1-1-2005

3D-SoftChip: A novel 3D vertically integrated adaptive computing system

Chul Kim
Edith Cowan University

Edith Cowan University

Copyright Warning

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

3D-SoftChip:

A Novel 3D Vertically Integrated Adaptive Computing System

A Dissertation

Presented to the School of Engineering and Mathematics

Edith Cowan University

Western Australia

In partial fulfillment of the requirements for the degree of

Master of Engineering Science

by

Chul KIM

Supervisor: ***Dr. Alexander Rassau***

Submission Date: ***June 2005***

EDITH COWAN UNIVERSITY

PERTH WESTERN AUSTRALIA

Dedication

To my fiancé Sang-Mi Hyun,

my father Nam-Gil Kim,

my mother Sung-Sun Park,

my brother-in-law Sun-Shin Lee,

and my sisters Hee-Joung, Su-Joung, Youn-Joung Kim.

Table of Contents

Declaration

I certify that this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education; and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature /

Date 12/06/05

Acknowledgements

I would like to express my gratitude to the following people, who helped me to stand this position.

Prof. Kamran Eshraghian as my principle supervisor who initiated the research program and gave me the opportunity to commence my master course at Edith Cowan University providing financial support and great inspiration towards my research. Unfortunately, he left the university towards the end of my research however he left significant impression of his great leadership that I want to follow.

Prof. Mike Myung-Ok Lee, who inspired me to study overseas and gave me an opportunity and warmth supervision during my course. I have learned strong propulsion and passion through his supervision.

Prof. Byung-Lok Cho, he used to be my supervisor during my undergraduate study. I have started with his great supervision and learned the life and belief as an electronic engineer. I will not forget his guidance that has changed my whole life.

Dr. Alexander Rassau, my principle supervisor, I am really a lucky fellow to meet him as a principle supervisor. Sometimes, he becomes my ear, mouth, hands and legs. I can not forget his infinite interest and supervision capacity for me. I could not finish my course without his great supervision which will never be forgotten and it is very much appreciated. Thanks Dr. Alexander Rassau.

My family is the most precious in my life. They motivated and encouraged me unfailingly so my deepest gratitude goes to my family and I dedicate my dissertation to my family; my father, my mother, my brother-in-law, my sisters and my future new family; my father-in-law, mother-in-law and my new brother-in-law.

Lastly, my fiancé Sang-Mi, she pushing me to study hard, but ironically, she gave me so many interruptions as well. But I even love these interruptions. I will promise that I will be a good husband and I will love you forever.

ABSTRACT

At present, as we enter the nano and giga-scaled integrated-circuit era, there are many system design challenges which must be overcome to resolve problems in current systems. The incredibly increased nonrecurring engineering (NRE) cost, abruptly shortened Time-to-Market (TTA) period and ever widening design productive gaps are good examples illustrating the problems in current systems. To cope with these problems, the concept of an Adaptive Computing System is becoming a critical technology for next generation computing systems. The other big problem is an explosion in the interconnection wire requirements in standard planar technology resulting from the very high data-bandwidth requirements demanded for real-time communications and multimedia signal processing. The concept of 3D-vertical integration of 2D planar chips becomes an attractive solution to combat the ever increasing interconnect wire requirements. As a result, this research proposes the concept of a novel 3D integrated adaptive computing system, which we term 3D-ACSoC. The architecture and advanced system design methodology of the proposed 3D-SoftChip as a forthcoming giga-scaled integrated circuit computing system has been introduced, along with high-level system modeling and functional verification in the early design stage using SystemC.

A major challenge in this research is to explore the proposed 3D-SoftChip platform to investigate the effectiveness of the first novel 3D vertically integrated Adaptive Computing System-on-Chip (ACSoC) as a next generation computing system. The suggested 3D-SoftChip has been modeled at a system level using SystemC and the functional verification of the modeled system has been firmly verified. The hand-crafted assembler code for implementation of the MPEG4 motion estimation algorithm has been applied with more than 3.8 times performance improvement over conventional systems. It can be clearly demonstrated that it is a highly suitable architecture for next generation computing systems. Finally, further work to realize the full implementation of the novel concept of a 3D-ACSoC has been suggested.

Publications

The following is a list of papers published during the course of this research.

International Journals

Chul Kim, Alex Rassau, Stefan Lachowicz, Mike Myung-ok Lee and Kamran Eshraghian (2005) – "***3D-SoftChip: A Novel Architecture for Next Generation Adaptive Computing Systems***", *to be published in EURASIP Journal on Applied Signal Processing.*

Chul Kim, Alex Rassau, Stefan Lachowicz, Mike Myung-ok Lee and Kamran Eshraghian (2005) – "***3D-SoftChip: A System-level Verification and Characterization of a Novel 3D Vertically Integrated Adaptive Computing System-on-Chip***", *Submitted in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*

International Conferences

Chul Kim, Mike Myung-ok Lee, Kamran Eshraghian and Byung Lok Cho (2004) – "***SoC-B Design and Testing Technique of IS-95C CDMA Transmitter for Measurement of Electronic Field Intensity using FPGA and ASIC***", *2nd IEEE International Workshop on Electronic Design, Test and Applications(DELTA2004), Perth, Australia, pp.251-254.*

Chul Kim, Mike Myung-ok Lee, Kamran Eshraghian and Byung Lok Cho (2005) – "***A Highly Accurate Electric Field Intensity Measurement System for IMT2000 and CDMA Network***", *Accepted on Advanced Industrial Conference on Telecommunication (AICT2005), Lisbon, Portugal.*

Chul Kim, Alex Rassau, Mike Myung-ok Lee and Kamran Eshraghian (2005) – "***3D-SoftChip: A Novel 3D Vertically Integrated Adaptive Computing System***", *13th ACM International Symposium on Field-Programmable Gate Array (FPGA2005), Monterey, California, U.S.A., Poster Section, pp.270*

Chul Kim, Alex Rassau, Mike Myung-ok Lee and Kamran Eshraghian (2005) – "***3D-SoftChip: A Novel 3D Vertically Integrated Adaptive Computing System***", *submitted in IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2005).*

Chul Kim, Alex Rassau, Mike Myung-ok Lee and Kamran Eshraghian(2005) – "***High-level System Modeling and Functional Verification of a 3D-SoftChip Adaptive Computing System using SystemC***", *submitted in IEEE International Conference on Computer Design (ICCD 2005)*

List of Figures

List of Tables

Chapter 1
Introduction

System design is becoming increasingly challenging as the complexity of integrated circuits and the time-to-market pressures relentlessly increase. Adaptive computing is a critical technology to develop for future computing systems in order to resolve most of the problems that system designers are now faced with due in no small part to its potential for wide applicability. Up until now, however, this concept has not been fully realized because of the many constraints such as chip real-estate limitations and the software complexity. Advancements of semiconductor processing technology and software technology, however, adaptive computing is now facing a turning point. For instance, the concept of reconfigurable computing has more recently started to receive considerable research attention [2, 3, 7] and this concept is now starting to move and expand into the realm of adaptive computing. Software defined virtual hardware [9] and "Do-it-all" devices [12] are good examples that demonstrate this development direction for computing systems.

Another growing problem in advanced computation systems, particularly for real-time communication or video processing applications, is the data bandwidth necessary to satisfy the processing requirements. A novel 3D integration system such as 3D SoC [24], 3D-SoftChip [14,15] which is able to satisfy the severe demand of more computation throughput by effectively manipulating the functionality of hardware primitives through

vertical integration of two 2D chips is another concept proposed for next generation computing systems. This research explores the proposed 3D-SoftChip platform to investigate the effectiveness of the first novel 3D vertically integrated Adaptive Computing System-on-Chip (3D-ACSoC) as a next generation computing system. This thesis outlines research into the system level design and functional verification of 3D-SoftChip in the initial stage of development of the novel 3D vertically integrated ACSoC.

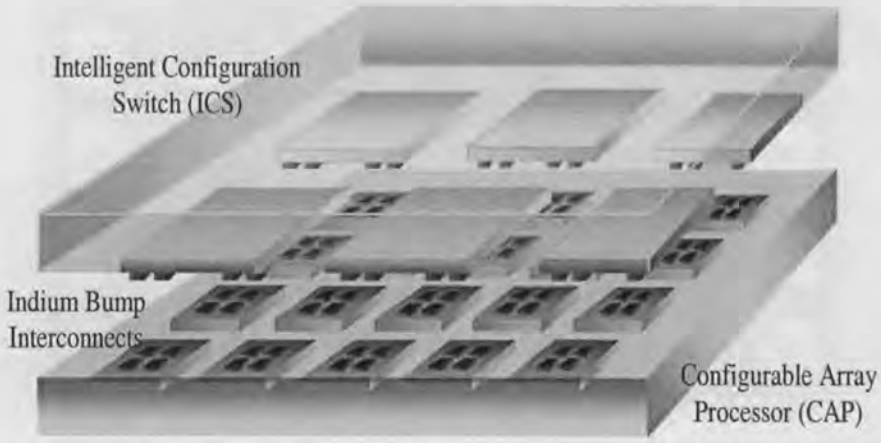


Figure 1.1: 3D-SoftChip Physical Architecture

Figure 1.1 illustrates the physical architecture of the 3D-SoftChip comprising the vertical integration of two 2D chips. The upper chip is the Intelligent Configurable Switch (ICS). The lower chip is the Configurable Array Processor (CAP). Interconnection between the two 2D chips is achieved via Indium bump interconnections. As the starting point for our 3D mapping, the 2-D plane architecture of the 3D-SoftChip is also illustrated in Figure 1.2 in order to demonstrate the principle.

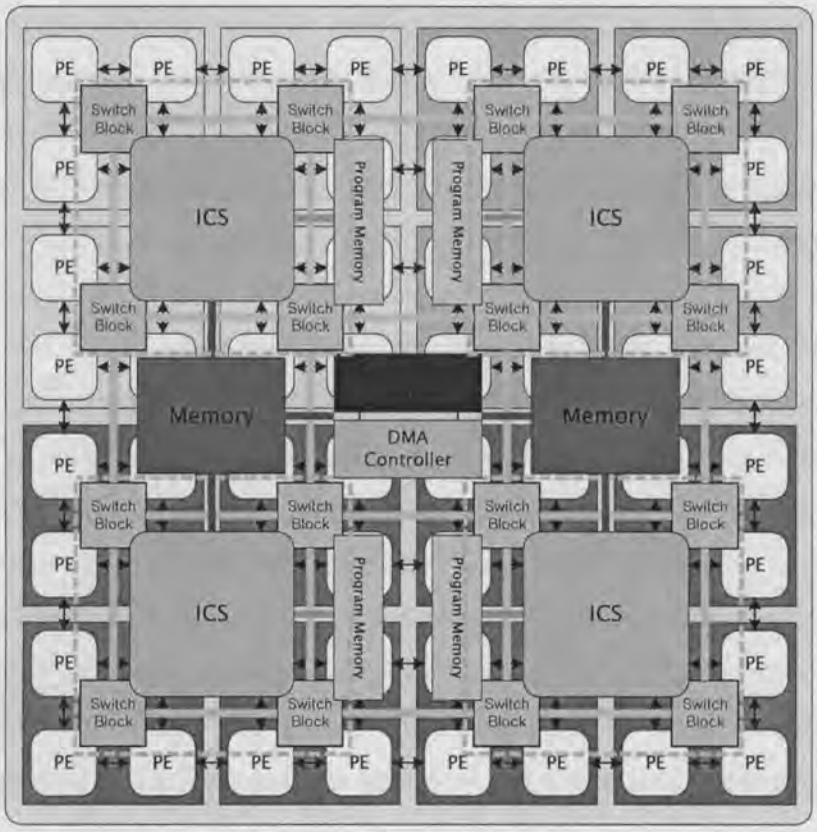


Figure 1.2: 3D-SoftChip: A novel 3D vertically integrated adaptive computing system-on-chip

1.1 3D Vertically Integrated Systems Overview

During the past few years, there has been significant research demand for 3D vertically integrated systems due to the ever growing wiring requirements, which are fast becoming the major bottleneck for future gigascale integrated systems [23,24]. In Very Deep Submicron silicon geometry, standard planar technology has many drawbacks such as performance, reliability etc. caused by limitations in the wiring. Moreover, the data bandwidth requirements for the next generation computing systems are becoming ever larger. To overcome these problems, the concept of 3D-SoC, 3D-SoftChip has been developed, which exploits the vertical integration of two or more 2D planar chips to effectively manipulate computation throughput. Previous work has shown that the 3D integration of systems can significantly reduce interconnection requirements [25]. As described by Joyner, et al [25], 3D system integration offers a 3.9 times increase in wire-limited clock frequency, an 84% decrease in wire-limited area or a 25% decrease in the

number of metal levels required per stratum. There are three feasible 3D integration methods; a stacking of packages, a stacking of ICs and Vertical System Integration as was introduced by IMEC [23]. There are four main enabling technologies for the fabrication of 3D-Integrated Circuits, Bean Recrystallization, Silicon Epitaxial Growth, Solid Phase Crystallization and Processed Wafer Bonding [26]. Table 1.1 shows the main characteristics of each of these 3D fabrication technologies. In this research, however, the focus is on the use of processed wafer bonding technology using an indium bump interconnection array (IBIA). The reason why wafer bonding technology is adopted for this work is because the process has particular benefits for applications where each chip carries out independent processing. The characteristic of the 3D-SoftChip are that each of the two planar chips should be effectively manipulated to maximize computation throughput with parallelism. Also indium has good adhesion, a low contact resistance and can be readily utilized to achieve an interconnect array with a pitch as low as 10μm. The development of the 3D integrated systems will allow improvements that should be seen in the packaging cost, the performance, the reliability and a reduction in the size of the chips.

Table 1.1: 3D Fabrication Technologies

3D Fabrication Technologies	Characteristics
Beam Recrystallization	Deposit poly-silicon and fabricate Thin-film Transistors (TFTs). High performance of TFT's High temperature of melting poly-silicon(Not practical Fab.Tech.) Suffers from low carrier mobility
Silicon Epitaxial (SE) Growth	Epitaxially grow a single crystal Si High temperature causes degradation in quality of devices Process not yet manufacturable
Solid Phase Crystallization	Low temperature alternative to SE Flexibility of creating multiple layers Compatible with current processing environments Useful for stacked SRAM and EEPROM cells
Processed Wafer Bonding	Bond two fully processed wafers together Similar electrical properties on all devices Independent of temperature since all chips are fabricated then bonded / Good for applications where chips do independent processing Lack of precision(alignment) restricts inter-chip communication to global metal line

1.2 Adaptive Computing Systems Overview

There are three types of computing system currently in existence; a general-purpose computing system, a reconfigurable/adaptive computing system and an application specific computing system. The general-purpose computing system is based on using a general-purpose processor for broad applications. Discrete application specific ICs are used for application specific computing systems for dedicated and limited applications. These computing systems have certain drawbacks such as low performance in the case of the general-purpose computing system, or extremely limited applicability for the application specific computing system. The reconfigurable/adaptive computing system, however, allows for an optimum trade-off between flexibility and performance. Because of this fact, reconfigurable/adaptive computing systems are attracting attention as a new alternative for the next generation of computing systems. Figure 1.3 illustrates how the reconfigurable/adaptive computing system provides an optimum trade off between flexibility and performance.

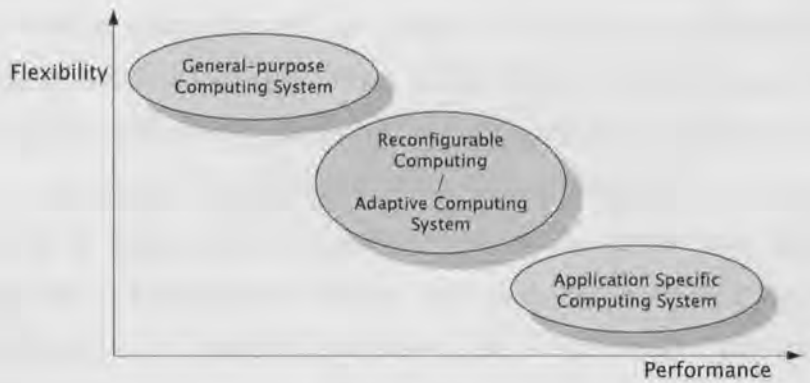


Figure 1.3: Computing systems

1.2.1 Adaptive Computing Systems

1.2.1.1 The Need for Adaptive Computing Systems

The nonrecurring engineering (NRE) costs associated with the design and testing of complex chips are one of the great threatening factors in current system design approaches. According to the International Technology Roadmap for Semiconductors

(ITRS), the manufacturing engineering costs of complex chips have reached almost one million dollars. The associated design NRE costs almost reached tens of millions of dollars in year 2003 [21]. Moreover, product life cycles are getting ever shorter due to rapid changes in technology and as a result the time-to-market (TTM) period is keenly shortened. On the other hand, design and verification cycle times are getting longer into the months or even years. As a consequence of these issues, a reconfigurable/adaptive computing system that could be metamorphosed across multiple standards and applications becomes very attractive for the next generation of computing systems.

1.2.1.2 The Concept of Adaptive Computing Systems

A reconfigurable system is one that has reconfigurable hardware resources that can be adapted to the application currently under execution providing the possibility to customize across multiple standards and applications. In most of the previous research the concepts of reconfigurable and adaptive computing have been described interchangeably. In this document, however, these two concepts will be more specifically described and differentiated. Adaptive computing will be treated as a more extended and advanced concept of reconfigurable computing systems, which means it includes more advanced software technology to effectively manipulate the mapping and scheduling of context memory over a wide range of applications along with more advanced reconfigurable hardware resources to support fast and seamless execution across these applications. Table 1.2 shows the differentiations between reconfigurable computing and adaptive computing. The benefits of adaptive computing are silicon reuse, bug-fixing post-shipping, updating and fixing in market allowing for standards evolution, faster TTM and lower costs. The reconfiguration capacity allows for significant reuse of silicon. If bugs are found post-shipping or standards evolve, the adaptive computing system is easy to fix and update simply by changing the contexts in the reconfigurable hardware resource. The forthcoming impact from the deployment of adaptive computing is "Do-it-all" devices. A small handheld PDA size device can assume the functionality of about 10 standard devices simply depending on the context programs included such as a cellular phone, a GPS receiver, an MP3 player, an e-book reader, a digital camera, a portable television, a

satellite radio, a held-held gaming platform etc. Figure 1.4 shows the futuristic concept of "Do-it-all" devices.

Figure 1.4: An Example of "Do-it-all" Device (*Source: www.chosun.com)

Table 1.2: Reconfigurable Computing Vs Adaptive Computing.

	Reconfigurable Computing	Adaptive Computing
Hardware Resources	Linear array of homogeneous elements (Logic gates, look-up tables)	Heterogeneous algorithmic elements (Complete function units such as ALU, Multiplier)
Configuration	Static, Dynamic configuration Slow reconfiguration time	Dynamic, partial run-time reconfiguration.
Mapping methods	Manual routing , conventional ASIC Design tools (HDL)	High-level language (SystemC,C)
Characteristics	Large silicon area, Low speed (high capacitance), high power consumption, high cost	Smaller silicon size, high speed, high performance, low power consumption, low cost

1.2.2. Classification of Adaptive Computing Systems

Adaptive computing systems are mainly classified in terms of granularity, programmability, reconfigurability, computational methods, hardware mapping methods and target applications. The granularity is the basic data size of the reconfigurable hardware resources. In fine grained systems, the primitive reconfigurable hardware resources are typically logic gates, flip-flops and look-up tables and operate using bit-level computations. Field Programmable Gate Array (FPGA) and Complex Programmable Logic Gates (CPLD) are good examples of fine grained systems. In contrast, the coarse grained systems have complete function units such as ALU, multiplier and dedicated functional units and operate using word-level computations. The combination of the fined grained systems and the coarse grained systems creates a mixed grained system.

The programmability relates to the capacity of the configuration. Single-programmability allows only one customization, while multiple-programmability allows for customization on-the-fly. The reconfigurability is executed by changing the context memory. Static (interrupted execution) and Dynamic (in parallel execution) are two categories of reconfigurability. Common computational methods used in the adaptive computing systems are Single-Instruction stream Multiple-Data stream (SIMD)/Multiple-Instruction stream Multiple-Data stream (MIMD) and Very-Long Instruction Word (VLIW). The hardware mapping methods vary depending on developed systems from manual routing to high-level language compilation. Most of the target applications for adaptive computing are in the areas of wired and wireless communications and multimedia digital signal processing

1.2.2.1 Previous Works

The research and commercial development of reconfigurable/adaptive computing systems has been going vigorously since the early 1990's. According to the classification of adaptive computing described above, the nature of this research is classified in the Table 1.3[3, 22] and it shows the best-known existing coarse-grain reconfigurable

systems, the fine-grain reconfigurable systems have been excluded because these are different category from our research.

The Matrix [1], REMARC [5] and MorphoSys [3] belong to the category of mesh-based reconfigurable systems which is a combination of an array of word-level processing elements with a control processor, such as a multi-granular array of Basic Functional Units (BFUs) in the case of the MATRIX, an 8 by 8 array of 16-bit nanoprocessor with MIPS-II RISC processor in the REMARC, or an 8 by 8 array of reconfigurable cells with MIPS-like processors in the MorphoSys. These are dynamic reconfiguration, mesh based hierarchical interconnection fabric architectures. Their application is restricted only to DSP type tasks and they have certain disadvantages in term of the power consumption because of frequent data movement between the control processor and processing elements. As well as, need to access external memory resources.

Another category is a linear array-based reconfigurable system such as, RaPiD [6] or PipeRench [2]. These are linear arrays of processing elements with row-wide interconnection fabrics. Each combination of the processing element array and the row-wide interconnection can make a pipeline stage. The target application of these systems is pipelining regular computation-intensive applications. The other categories such as crossbar-based [35, 36] and reconfigurable processors [37] have been excluded in this table.

The Trisend A7 [10] is considerably similar to other mesh-based reconfigurable systems, the difference is in the granularity of the processing elements. The A7 has a fine-grain reconfigurable fabric in comparison with the word-level processing elements in the mesh-based reconfigurable system.

The MRC6011 [11], Adapt 2400 [16], DFA1000 [9], PC102 [17] are up-to-date commercially developed adaptive computing systems, which have mostly heterogeneous arrays of reconfigurable hardware except the DFA1000 and dynamic configurability. The main target application is computation-intensive multimedia DSP and communication signal processing. These have more advanced adaptive computing characteristics compared with the systems introduced earlier.

As indicated, the early research and development was into single linear array type reconfigurable systems with single and static configuration [8,1,6,5,4,2] but this has evolved to large adaptive SoCs with heterogeneous types of reconfigurable hardware resources and multiple and dynamic configurability. The MRC6011, Adapt2400, DFA100, PC102 and 3D-SoftChip are good examples to show the current research and commercial development directions. The ultimate goal for the adaptive computing system is currently the "Do-it-all" device as explained before.

Table1.3: Reconfigurable and Adaptive Computing Systems

System	Granularity	Programmability	Reconfiguration	Computation Method	Mapping	Target Application
PADDI [8]	Coarse(16bit)	Multiple	Static	VLIW, SIMD	Routing	DSP applications
MATRIX [1]	Coarse(8bit)	Multiple	Dynamic	MIMD	Multi-length	General Purpose
RaPiD [6]	Coarse(16bit)	Single	Mostly static	Linear array	Channel routing	Systolic arrays, Data-intensive
REMARC [5]	Coarse(16bit)	Multiple	Static	SIMD	N/A	Data-parallel application
RAW [4]	Mixed	Single	Static	MIMD	Switch box routing	General purpose
PipeRench [2]	Mixed(128bit)	Multiple	Dynamic	Pipe-lined	Scheduling	Data-parallel, DSP applications
MorphoSys [3]	Coarse(16bit)	Multiple	Dynamic	SIMD	Assembler, Manual P&R	Data-parallel, Image applications
Triscend A7 [10]	Mixed	Multiple	Dynamic	N/A	Co-compilation (Assembler, C, Verilog, VHDL)	General Purpose Embedded System
Motorola MRC6011 [11]	Coarse(16bit)	Multiple	Dynamic	SIMD	C-Compilation	Computation Intensive applications
QuickSilver Adapt2400 [16]	Coarse(8,16,24, 32bit)	Multiple	Dynamic	Heterogeneous Nodes array	SilverC	Comm., Multimedia DSP
Elixent DFA1000 [9]	Coarse (4bit)	Multiple	Dynamic	Linear D-Fabric Array	Verilog, VHDL, Handle-C, Matlab	Multimedia applications
picoChip PC102 [17]	Coarse(16bit)	Multiple	Dynamic	3way-LIW	Assembler	Wireless Communications
3D-SoftChip	**Coarse(4bit)**	**Multiple**	**Dynamic**	**Various types of computation models**	**C-compilation (Assembler, C)**	**Comm. Multimedia Signal Processing**

1.2.2.2 MorphoSys Vs 3D-SoftChip

One of the most successful reconfigurable systems to date is the MorphoSys system, so it is meaningful to make a comparison of the proposed 3D-SoftChip architecture to this. Table 1.4 shows the comparison between the MorphoSys and the 3D-SoftChip. It can be seen that the 3D-SoftChip is more appropriate to the most up-to-date adaptive computing system.

Table1.4: Comparison of MorphoSys with 3D-SoftChip

	MorphoSys	3D-SoftChip
Integrated Model	System-on-Chip except main memory	Vertically Integrated complete System-on-Chip with abundant memory capacity
Memory Interface	Employs a two-set data buffer that enable overlap of computation with data transfers.	Using Indium bump technology, vertical data communication. Variable memory word-length for adaptive computing
Reconfiguration	Multiple contexts on-chip (32 planes) with dynamic and single-cycle.	Multiple context on-chip with dynamic and single-cycle
Controller	On-chip general-purpose processor.	Every unit 3D-SoftChip has an ICS_RISC which role of control processor.
Examples of Application	MPEG-2 Video Compression, Encoder Automatic Target Recognition Data Encryption	Real time communication and multimedia signal processing
Characters	SIMD nature. Fixed Word length Comprehensive tool sets.(mView, mLoad, mSched, mcc, MuLate, MorphoSim)	Various types of Computational models (SISD,SIMD,MISD,MIMD) And 3 types of SIMD Computation models(massively parallel, multithreaded, pipelined) Configurable word length and variable memory word length for Adaptive Computing. 3D Vertically Integrated System – High speed data interface. Optimum System Architecture for Comm. and Multimedia Signal Processing

1.3 Motivation of Thesis

As the microelectronics industry enters the nano and giga-scaled integrated circuit era, many problems, as described before, have been to occur. To cope with these problems, especially the system-on-chip complexity and interconnection crisis, innovative new computing systems with novel interconnection methods will be required. A very promising candidate to overcome these problems is the concept of a 3D vertically integrated adaptive computing system-on-chip (3D-ACSoC). This concept may well be a critical technology for the next generation of computing systems because of its wide applicability/adaptability and because of the significant benefits gained from 3D systems such as reduction in interconnect delays and densities, and reduction in chip areas due to the possibility for more efficient layouts etc.

Conventional SoC design methodologies include many error-prone and tedious iteration processes which can result in a lack of system reliability and extend the design time. Moreover, the portion taken up by verification processes in the total design time is exponentially increasing. By adopting the suggested SoC design methodology using SystemC, the design time can be significantly reduced and more reliable systems can be realised. To satisfy these needs, the concept of the 3D-ACSoC and advanced HW/SW co-design and verification methodology has been suggested.

1.4 Scope of Thesis

In this thesis, the novel 3D-SoftChip architecture for real-time communication and multimedia signal processing is introduced, and its high-level system modelling and functional verification using SystemC is described. The 3D-SoftChip has been fully modelled using SystemC at high-level and implementation of the MPEG4 full search block matching motion estimation algorithm has been mapped to the modelled 3D-SoftChip. Finally, the performance analysis is detailed in the last chapter. The thesis is composed of nine chapters including this one. The following is the scope to be covered by each of the following chapters.

1.4.1. Scope of Each Chapters

Chapter 2 is an introduction to the overall 3D-SoftChip architecture. The novel architecture and several salient features for next generation computing system will be introduced along with the suggested HW/SW co-design and verification methodology. The detailed architecture of the CAP chip will be described in Chapter 3. Heterogeneous types of Processing Elements architecture and functions will be presented. Chapter 4 covers the ICS chip, its components and the ICS_RISC instruction Set Architecture with instruction set summary. Chapter 5 presents the architecture of the UnitChip, its pipeline operation mechanism and area constraint. A three hierarchical interconnection architecture and the configurable nature of the inter-PE bus will be introduced in Chapter 6. Chapter 7 presents the high-level modelling of 3D-SoftChip using SystemC. The simulation result of each component of the 3D-SoftChip is also provided to show the verification of the functionality of the each component. Chapter 8 introduces application mapping for high-level modelled 3D-SoftChip with the MPEG4 full search block matching algorithm. The performance analysis will be performed in comparison with conventional systems. Finally, the last Chapter outlines the contributions of this thesis and suggested future work.

1.5 Conclusions

In this chapter, the motivation for the emergence of the novel 3D vertically integrated system-on-chip and the benefits which can be acquired through its use have been described. This concept will be a promising candidate for the next generation of computing system.

Chapter 2
System Architecture of 3D-SoftChip

In this chapter, the core technology for the 3D-SoftChip along with its detailed and overall architecture will be described. Finally a design guideline and the suggested design methodology will be introduced.

2.1 Core Technology for 3D-SoftChip

The core technology for the 3D-SoftChip can be mainly classified into 3 fields of technology is follows, a Very Deep Submicron (VDSM) silicon process technology, a 3D Interconnection technology and an advanced software technology. The target silicon process technology for the 3D-SoftChip is less than 0.13um to maximize the effect of large scale integration in order to fit as much as possible into the Processing Element (PE)s. This large scale integration into the PEs can be leveraged to amplify the computation capacity of the 3D-SoftChip because of the SIMD computation nature of the 3D-SoftChip. The 3D Interconnection technology using the Indium Bump Interconnection Array (IBIA) is another state-of the art technology for the 3D-SoftChip. The IBIA can cope with the severe demand for data bandwidth from real time communication and multimedia signal processing applications. The last technology is the advanced software technology, which is able to effectively execute context mapping and scheduling within the context memory in the 3D-SoftChip.

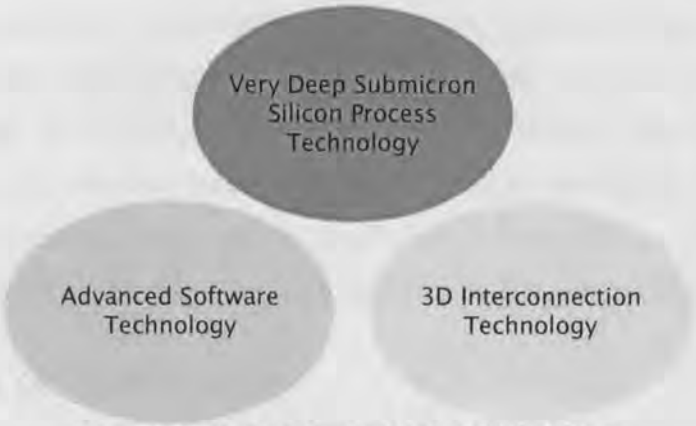


Figure 2.1: Core technology for the 3D-SoftChip

2.2 Overall Architecture of 3D-SoftChip

Figure 2.2 shows the overall architecture of the 3D-SoftChip.

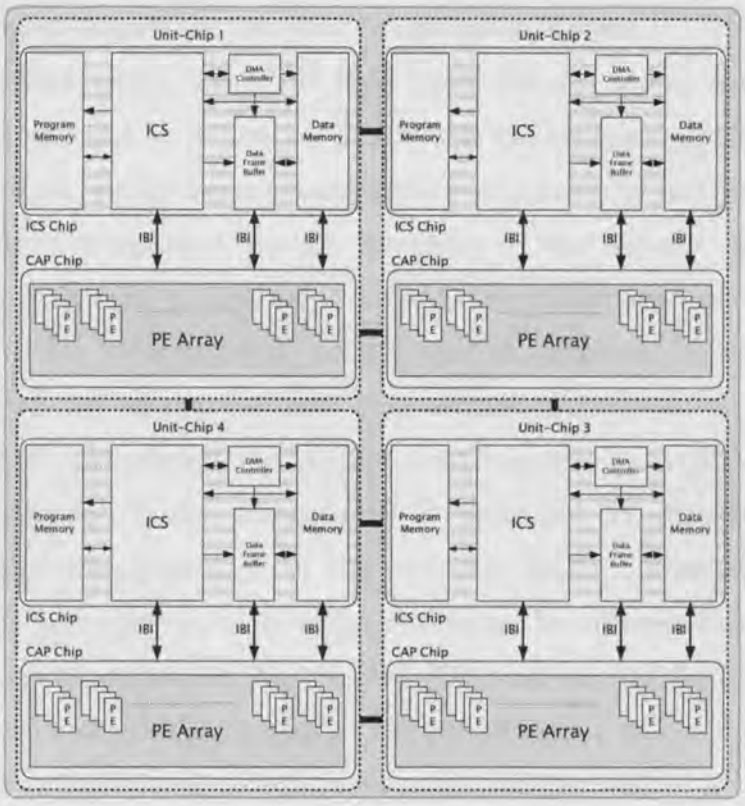


Figure 2.2: Overall architecture of the 3D-SoftChip

As can be seen, it is comprised of 4 UnitChips. By including four separate unit chips in the architecture, sufficient flexibility is provided to allow multiple optimized task

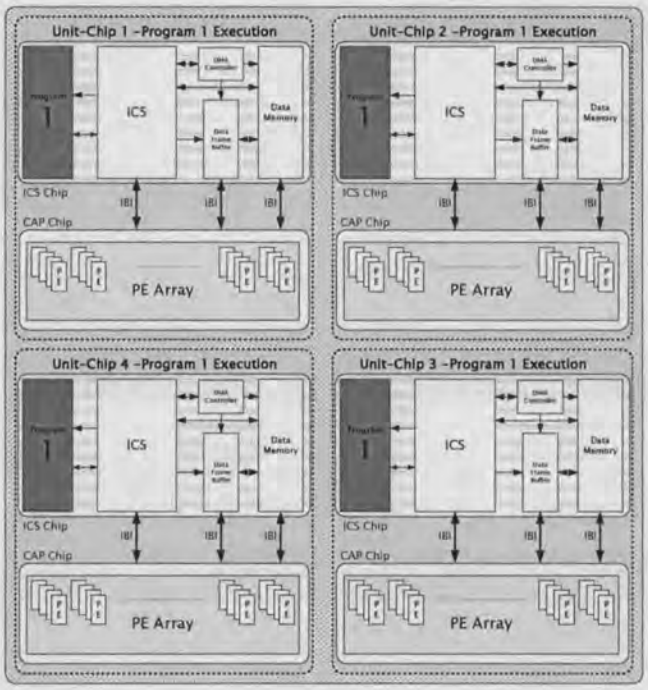
threads to be processed simultaneously. Given the primary target applications of communication and multimedia processing four UnitChips should be sufficient for all such requirements. Each UnitChip has a PE array, a dedicated control processor and a high bandwidth data interface unit. According to a given application program, the PE array processes a large amount of data in parallel, the ICS controls the overall system and directs the PE array execution and data and address transfers within the system.

2.3 Features of 3D-SoftChip

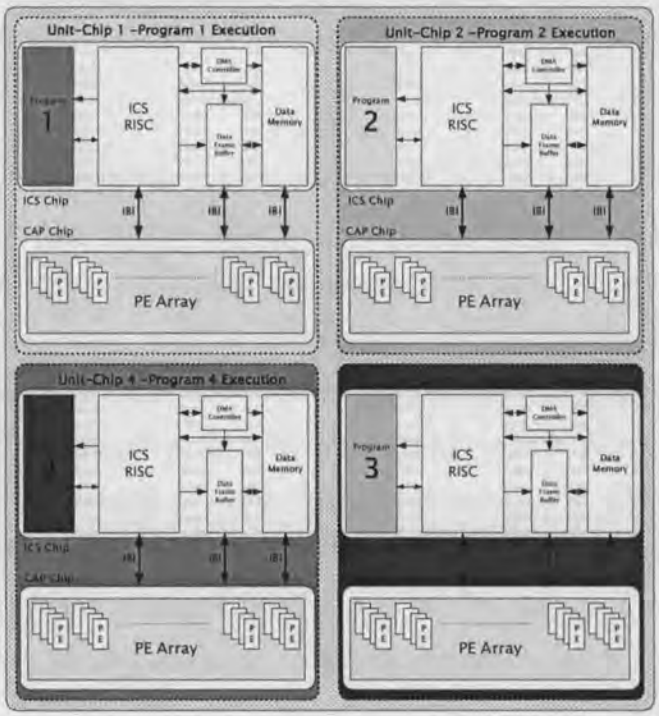
The 3D-SoftChip has 4 distinctive features: Various types of computation modes, adaptive Word-length configuration [14], optimized system architecture for real-time communication and multimedia signal processing and dynamic reconfigurability for adaptive computing.

- **Computation Algorithm : Various Computation Models**

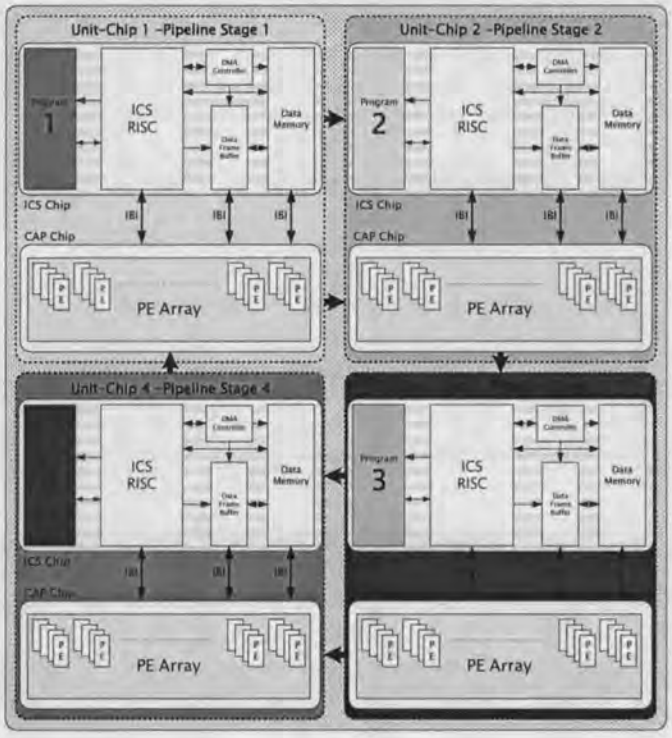
  As described above, one 32-bit RISC controller can supply control, data and instruction addresses to 16 sets of PEs through the completely freely controllable switch block so various computation models can be achieved such as SISD, SIMD, MISD, MIMD as required. Enough flexibility is thus achieved for an adaptive computing (AC) system. Especially, in the SIMD computation model, 3 types of different SIMD computational model can be realized, massively parallel, multithreaded and pipelined SIMD computational models [13]. In the massively parallel SIMD computation model, each UnitChip operates with the same global program memory. Every computation is processed in parallel, maximizing computational throughput. In the Multithreaded SIMD computation model, the executed program instructions in each UnitChip can be different from the others, so multithreaded programs can be executed. The final one is the pipelined SIMD computation model. In this case each UnitChip executes a different pipelined stage. These three computational models are illustrated in Figure 2.3.



(a) Massively Parallel SIMD Computational Model



(b) Multithreaded SIMD Computational Model

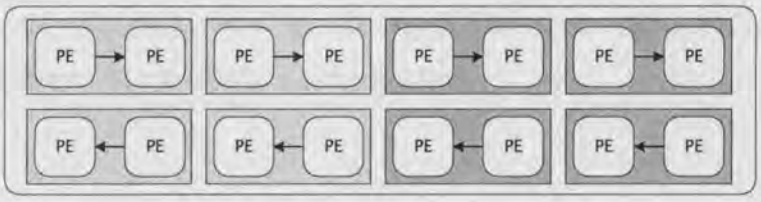


(c) Pipelined SIMD Computation Model

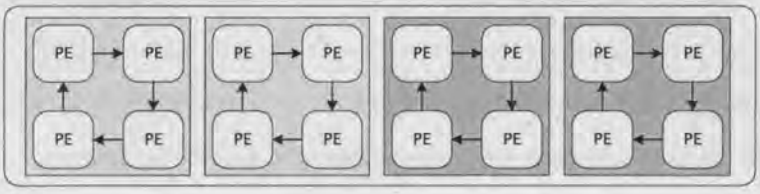
Figure 2.3: Computation Algorithm: 3 types of SIMD Computation Models

- **Word-length Configuration**

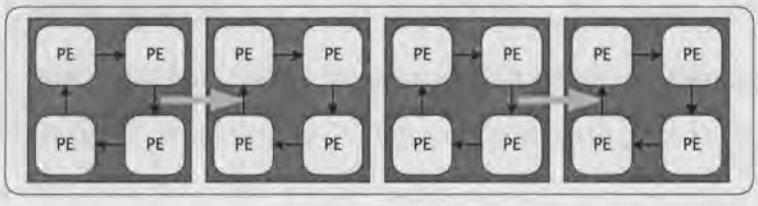
This is a key characteristic in order to classify the 3D-SoftChip as an adaptive computing system. Each PE's basic processing word-length is 4-bit. This can, however, be configured up to 32-bit according to the application in the program memory. Figure 2.4 illustrates the proposed word-length configuration algorithm. When 2 PEs configure together, an 8-bit word-length system is created. If 4 PEs configure together this extends to 16-bit. And finally when 8 PEs configure together a full 32-bit word length is achieved. This flexibility is possible due to the configurable nature of the arithmetic primitives in the PEs [18], (see chapter3.5) and the completely freely controllable switch block architecture in the ICS chip



(a) 8-bit Word-length Configuration



(b) 16-bit Word-length Configuration



(c) 32-bit Word-length Configuration

Figure 2.4: Word-length Configuration Algorithm

- **Optimized System architecture for Communication and Multimedia Signal Processing**

There are many similarities between communications and multimedia signal processing, such as data parallelism, low precision data and high computation rates. The different characteristics of communication signal processing are basically more data reorganization such as matrix transposition and potentially higher bit level computation. To fulfill these signal processing demands, each UnitChip contains two types of PE. One is a standard-PE for generic ALU functions, which is optimized for bit-level computation. The other is a processing accelerator-PE for Digital Signal Processing (DSP). In addition, special addressing modes to leverage the localized memory along with 16 sets of Loop buffers to generate iterative

address in the ICS_RISC add to the specialized characteristics for optimized communication and multimedia signal processing.

- **Dynamic Reconfigurability for Adaptive Computing**

  Every PE contains a small quantity of local embedded SRAM memory and additionally the ICS chip has an abundant memory capacity directly addressable from the PEs. With multiple sets of program memory and the abundant memory capacity, it is possible to switch programs easily and seamlessly, even at run-time.

2.4 System Components

As introduced above, the 3D-SoftChip consists of a linear array of heterogeneous PEs with an associated array of Indium bump 3D Interconnects, dedicated Switch Blocks, the ICS_RISC and a high bandwidth data interface unit.

2.4.1. Configurable Array Processor (CAP) Chip

2.4.1.1 Heterogeneous Types of PEs

The CAP chip comprises a linear array of two types of PE, a Standard-PE and a Processing Accelerator-PE. The advantages of heterogeneous PEs with dedicated functions for special purpose DSP are more suitability for specific applications with only a medium flexibility trade-off compared with homogeneous type PEs. In this case, two Standard-PE and two Processing Accelerator-PEs form one Quad-PE. These will be in detail in a later section.

2.4.2. Intelligent Configurable Switch (ICS) Chip

2.4.2.1 Switch Block

Each group of 4 PEs (Called Quad-PEs) are controlled by one Switch Block through the IBIA. This transfers data from/to each PE and also provides instruction data for the

PEs. It can completely freely configure each PE group, and makes it possible to achieve efficient variable word-length configuration.

2.4.2.2 ICS_RISC

A 32-bit dedicated RISC processor is used to control each set of 4 Quad-PEs (called UnitCAP). It controls the execution of the PE array and provides control and address signals to the Switch Block and the high bandwidth data interface unit in the UnitChip.

2.4.2.3 Data Frame Buffer

Two sets of Data Frame Buffers are included to support the transfer of large volumes of data from/to data/program memory and the ISC.

2.4.2.4 Program Memory

This is separated into two areas. One is a program memory for the ICS_RISC and the other is the program memory for the PE array. This memory supports adaptively configured word-lengths to increase the computation efficiency dependent on the application. Additionally, multiple sets of program memory are included to allow dynamic program switching.

2.4.2.5 Data Memory

Abundant memory capacity is one of the characteristic of the 3D-Softchip with each PE containing its own embedded local memory along with a high bandwidth connection to the memory store on the ICS.

2.4.2.6 DMA Controller

A dedicated controller is included to facilitate the transfer of large volumes of data from/to program memory, data memory and the ICS. This provides a high efficiency data interface between any of these units.

2.4.2.7 3D Interconnection Technology

The CAP chip carries out all data manipulation operations in the system. There is rarely the need for data transfer within the CAP beyond basic nearest neighbor interconnects, except for computation with word-lengths configured to > 4-bit. All the manipulated data is, therefore, transferred through the Indium Bump Interconnection Array (IBIA) and processed by the ICS allowing for very high speed computation because the IBIA provides very high bandwidth and very low inductance/capacitance [15].

2.5 Design Guidelines

The design guidelines and constraints to satisfy the design goals are as follows.

- The 3D-SoftChip is the first novel 3D vertically integrated Adaptive Computing System-on-Chip (3D-ACSoC)
- Using Indium bump technology, data can be manipulated at very high speed with wide bandwidth.
- The variable memory word-length and configurable word-length are unique features for an adaptive computing system.
- Various computation models (SISD, SIMD, MISD, MIMD) are possible for adaptability/flexibility in accordance with the current application and 3 types of SIMD computational models (massively parallel, multithreaded, pipelined) allow for maximized computational throughput.
- The heterogeneous types of PE architectures are optimized for communication and multimedia signal processing.
- Dynamic run-time reconfigurability for adaptive computing. (Multiple sets of program memory and abundant memory capacity)
- The area constraint of PE should be minimized as much as possible (less than 60um × 60um in 0.13um technology) for a 4-bit word size.

2.6 Design Methodology

2.6.1. Suggested HW/SW Co-design and Verification Methodology

HW/SW co-design is a development methodology that supports the concurrent and co-operative development of hardware and software (co-specification, co-development, co-verification). It helps to evaluate the effect of design decisions and to explore the design space at an early stage to obtain the optimal architecture. As a result of this, design cost and design cycle time can be reduced and more reliable system can be realised because of the verification at the high-level of the system. Figure 2.5 shows a suggested HW/SW co-design methodology for the 3D-SoftChip. Once the system specification is firmly decided, HW/SW partitioning is executed to determine which functions should be implemented in hardware and which in software. The HW can then be modeled using SystemC [19] and SW modeled in C. After that, a co-simulation and verification process is implemented to verify the 3D-SoftChip operation and performance and to decide on an optimal HW/SW architecture.

More specifically, the SW is modelled using a modified GNU C Compiler and Assembler. After the compiler and assembler for ICS_RISC has been finalised, a program for the implementation of the MPEG4 motion estimation algorithm will be developed and compiled using it. After that, object code can be produced, which can be directly used as the input stimulus for an instruction set simulator and system level simulation. The HW/SW verification process can be achieved through the comparison between the results from instruction level simulation and system level simulation. From this point on, the rest of the procedure can be processed using any conventional HW design methodology, such as full and semi-custom design. N.B. SystemC is a system design language which supports concurrent HW/SW co-design methodologies and offers a simulation kernel that supports hardware modeling concepts at the system level, behavioral level and register transfer level [20].

2.7 Conclusions

The core technology, overall and detailed architecture of the 3D-SoftChip has been presented. The four kinds of salient features, as described Section 2.3 can differentiate the 3D-SoftChip from conventional reconfigurable/adaptive computing systems. The design time and reliability of the system will be significantly improved by adopting the suggested HW/SW co-design and verification methodology using SystemC.

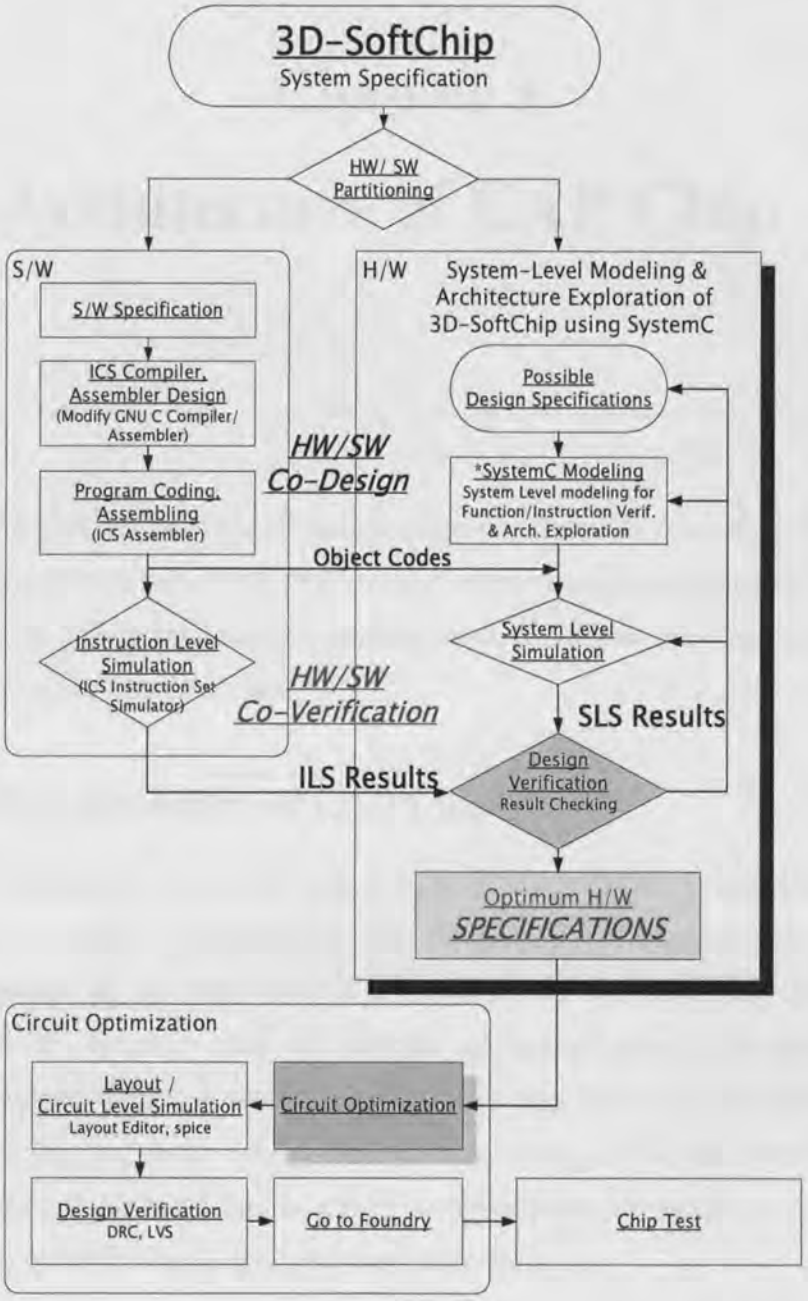


Figure 2.5: Suggested HW/SW Co-design and Verification Methodology

Chapter 3
Architecture of CAP Chip

In this chapter, the overall architecture of the Configurable Array Processor (CAP) chip will be described along with the PE architecture for communication and multimedia signal processing. The integration of 4 heterogeneous PEs forms one Quad-PE, and four Quad-PE make up the UnitCAP chip.

3.1 Overall Architecture of CAP Chip

The basic architecture of the CAP chip is a linear array of heterogeneous PEs. Figure 3.1 shows three possible architecture choices for the PEs. The architecture in Figure 3.1(b) is suggested as the most feasible architecture for the PE in the 3D-SoftChip because it has the optimum trade-off between application specific performance and flexibility. Examples of type A can be seen in [1,2,3], type B in [16] and type C in [17]. The CAP chip has the basic role of the processing engine for the 3D-SoftChip. It manipulates large amounts of data at a high computational rate using any of the three different SIMD computation models previously described.

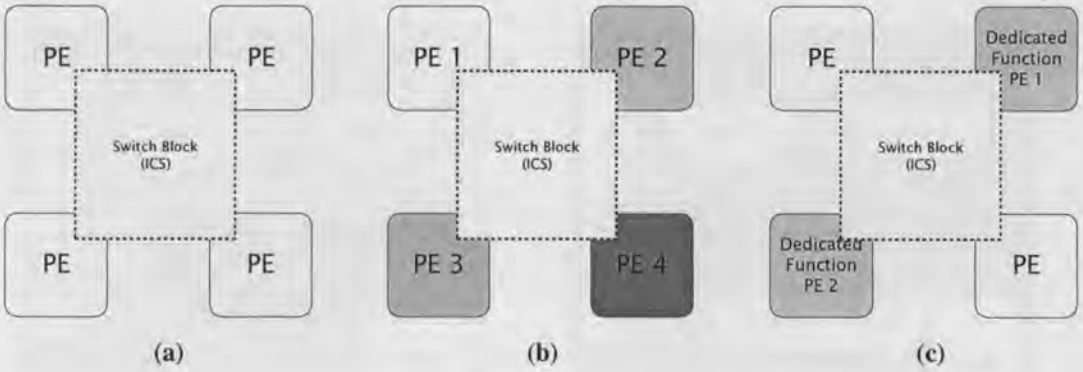


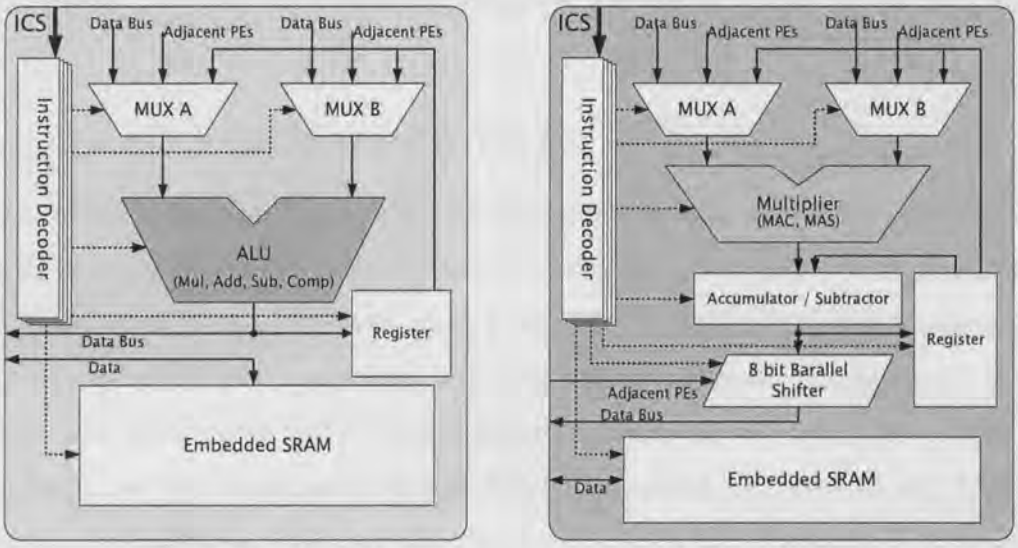
Figure 3.1: Types of PEs (a) homogeneous type, (b) heterogeneous type, (c) heterogeneous type with dedicated functions for special purpose.

Table 3.1: Characteristics of each PE types

	PE Architecture	Flexibility	Performance
Type A	Homogeneous type PEs with Embedded memory, ALU, MAC, Address decoder etc.	Suitable for general purpose High flexibility	Relative low performance for specific applications.
Type B	Each PEs are optimized for special functions Example : •PE1: Multiple MAC, ALU array •PE2: Bit-oriented operations •PE3: General purpose RISC or Control Logic. •PE4: Memory	Suitable for specific applications. Medium flexibility	Relative medium performance for specific applications
Type C	Combination of the Type B arch. with dedicated functions for special purpose Example : •PE1: Multiple MAC, ALU array •PE2: Memory and Control •PE3,PE4: A Co-processor optimized for dedicated signal processing functions (FEC, Preamble detect etc)	Suitable for dedicated applications. Low flexibility	Relative high performance for specific applications

3.2 Two Types of Processing Element (PE)s

Figure 3.2 illustrates the two type of PE architecture chosen to optimize communication and multimedia signal processing type applications. Table 3.2 shows the characteristics of the two type of PE.



(a) Standard-PE **(b) Processing Accelerator-PE**

Figure 3.2: Two Types of PE

Table 3.2: Characteristics of the two type of PE

	Standard-PE	Processing Accelator-PE
Components	Standard ALU(Mul, Add, Sub, Comparator) MUX A,B, Registers, Embedded SRAM	Multiplier, modified Adder, Subtractor, 8-bit Barrel Shifter, Registers, Embedded SRAM
Purpose	Bit-wise manipulation, Standard ALU functions	Dedicated for MAC, MAS functions (for DSP application)
Characteristics	Standard ALU functions Comparison operation.	Single clock cycle MAC, MAS absolute value computation operations 8bit barrel shifter. (Logical, Arithmetical Shift)

3.2.1. Standard-PE (S-PE)

The S-PE is for standard ALU functions and is also optimized for bit-level operation for communication signal processing. It comprise 4 sets of 19-bit registers for S-PE instruction decoding, two multiplexers to select input operands from the data bus, adjacent PEs or internal registers, a standard ALU with bit-serial multiplier, adder, subtractor and comparator, embedded local SRAM and 4 sets of Registers. The arithmetic primitives are scalable so as to make it possible to reconfigure the word-length for

specific tasks. The scalable architecture arithmetic primitive architecture is presented in [18].

3.2.2. Processing Accelerator-PE (PA-PE)

The PA-PE is dedicated specifically for Digital Signal Processing (DSP) operations. It consists of 4 sets of 19-bit registers for PA-PE instruction decoding, two multiplexers to select input operands from the data bus, adjacent PEs or internal registers, a signed 4-bit scalable parallel/parallel multiplier, and accumulator/subtractor modified to enable Multiple-and-Accumulate (MAC), Multiple-and-Subtract (MAS) operations within one clock cycle, an 8-bit configurable barrel shifter, embedded local SRAM and 4 sets of Registers. Two shifters in the Quad-PE can also be configured to produce a 16-bit barrel shifter. Its distinctive features are the single clock cycle MAC, MAS operations and parallel/parallel multiplier to accelerate DSP applications. Moreover it can execute single clock cycle absolute value computation.

3.3 PE Functions

PE functions are mainly divided into S-PE or PA-PE functions.

3.3.1. Standard-PE Functions

Table 3.3 shows the functions of S-PE. It is useful for bit-wise manipulation and generic ALU functions.

Table 3.3: Standard-PE Functions

Function	Mnemonics
A and B	AND
A or B	OR
not A	NOT
A xor B	XOR
A + B	ADD
A – B	SUB
A × B	SPMUL
A comp B	COMP

3.3.2. Processing Accelerator-PE Functions

Table 3.4 describes the PA-PE functions. It is specialized for DSP such as MAC, MAS, logical Shift, Arithmetic Shift, Rotate function, absolute value computation.

Table 3.4: Processing Accelerator-PE Functions

Function	Mnemonics
A × B	PAMUL
A × B + out(t)	MAC
A × B – out(t)	MAS
Logical Shift Left	LSL
Logical Shift Right	LSR
Arithmetic Shift Right	ASR
Rotate	ROR
|A|(Absolute value)	ABS

3.3.3. PE Instruction Formats and Operation Modes

The PE instruction format consists of a 19-bit instruction word. The most significant 2-bits, 18 and 17 in the instruction word (WS_en/RS_en, WR_en/RR_en) are used for the Read/Write enable bit of the embedded SRAM and registers. Bits 16 to 10 are used for SRAM and register selection (addressing). Bit 9 is used for data output register enable signal and bits 8 to 6 are used to specify the PE operation. Finally, bits 5 to 0 are used to control the input multiplexers for input operand selection. This format is illustrated in Figure 3.3 below.

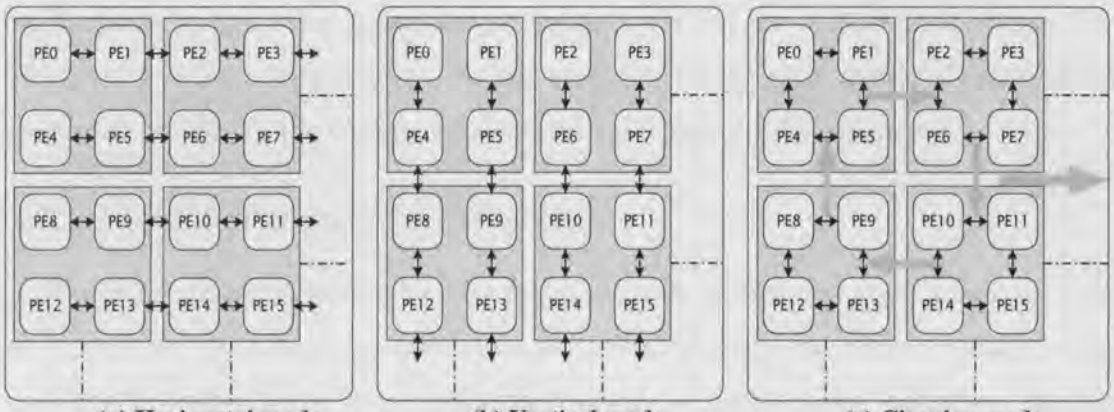
18	17	16	15 12	11 10	9	8 6	5 3	2 0
WS_en/ RS_en	WR_en/ RR_en	SRAM en	SRAM Selection	Register Selection	Dout RCtl	SPEOP	MUXB	MUXA

(a) Standard-PE Instruction format

18	17	16	15 12	11 10	9	8 6	5 3	2 0
WS_en/ RS_en	WR_en/ RR_en	SRAM en	SRAM Selection	Register Selection	Dout RCtl	PA-PE OP	MUXB	MUXA

(b) Processing Accelerator-PE Instruction format

Figure 3.3: PE Instruction formats



(a) Horizontal mode (b) Vertical mode (c) Circular mode

Figure 3.4: PE Array Operation Modes

Figure 3.4 illustrates 3 types of PE operation modes that can be realized on the PE array; Horizontal mode, Vertical mode and Circular mode. In the horizontal and vertical mode, the each rows or columns of the PEs can connected together respectively. These operation modes optimized for the SIMD computational method. Lastly, in the circular mode, the PEs in the one Quad-PE connects together and each Quad-PE can work separatively. These allow for even greater flexibility and help to maximize computational throughput according to the target application.

3.4 Embedded Local SRAM

Each PE has a small quantity of local embedded SRAM. As the effective memory bandwidth is increased dramatically by as much as the number of the PEs, which will result in an increase in effective processing speed in many applications. Bus traffic can also be reduced because many data transmission operations can be contained within a PE. Consequently, a lowering of power dissipation will also be achieved. Effectively this can act as cache, which can be continuously refreshed

3.5 Configurable Nature of Arithmetic Primitives

As described in the Chapter 2.3, one of the distinguished features as an adaptive computing system is the word-length configuration. The basic word-length of each PE is 4-bit. It can be configured 8, 16, 32-bit according to the target application. The

configurable nature of the arithmetic primitives in the PE allows this configuration [18]. The most complex component in the PE is multiplier so the example of configurable arithmetic primitives, the configurable parallel multiplier will be introduced

3.5.1. Scalable Parallel Multiplier Cell

Figure 3.5 shows a generic 1×1-bit multiplier cell. It includes a full adder, an AND gate and three multiplexers to select the input operand through the control signals CTRLH and CTRLL. In this figure, A represents the multiplicand and B is the multiplier. SIN is the SUM signal from the adjacent cell above, COUT is the propagated carry output, CIN is the carry input from the adjacent multiplier cell. MOUT represent the multiplication result. The 2×2-bit multiplier can be implemented using the generic 1-bit cell and moreover an 8×8-bit multiplier can be realised by arranging the basic 4×4 primitive in a 2×2 array as shown in figure 3.6 [18]. Because of this configurable characteristic, the word-length can be extended up to 32-bit in the 3D-SoftChip

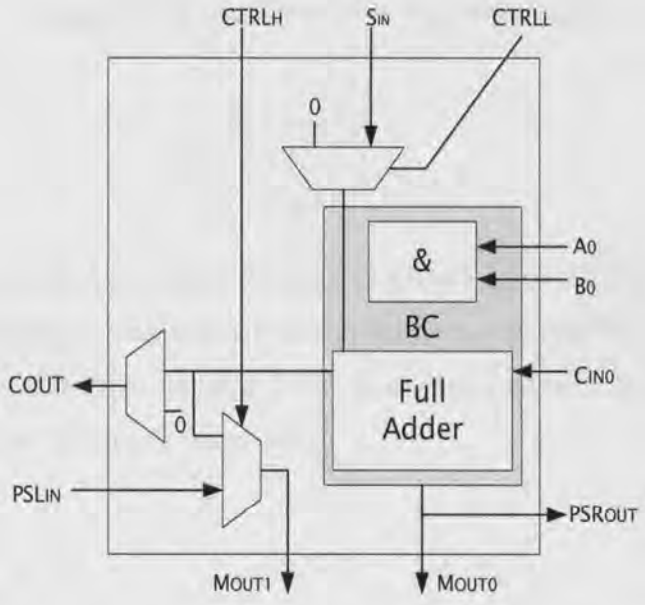


Figure 3.5: A generic 1×1-bit Multiplier Cell for n=1

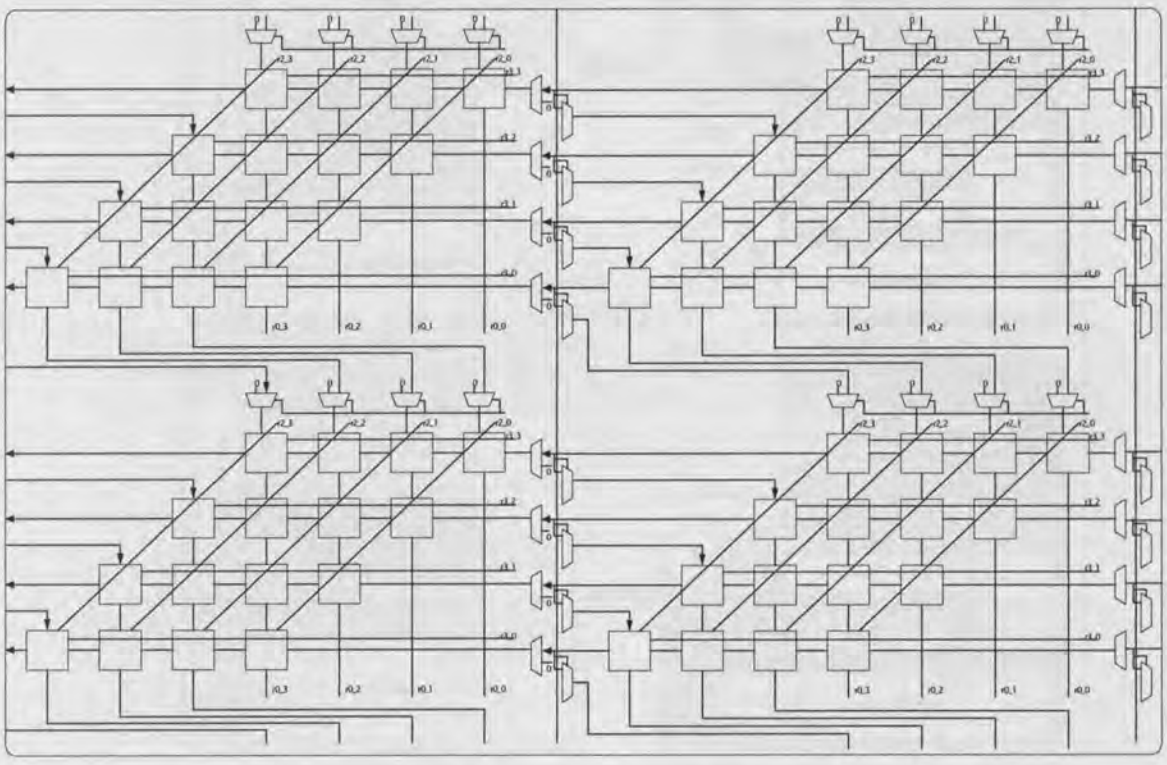


Figure 3.6: 8×8 multiplier using 4-bit Generic Cells

3.6 Quad-PE

As previously described one Quad-PE consists of two pairs of PEs (two S-PE and two PA-PE). The Quad-PE is controlled and configured by the Switch Block according to the control and address data from the ICS_RISC transmitted through the IBIA. Figure 3.7 shows the architecture of a single Quad-PE.

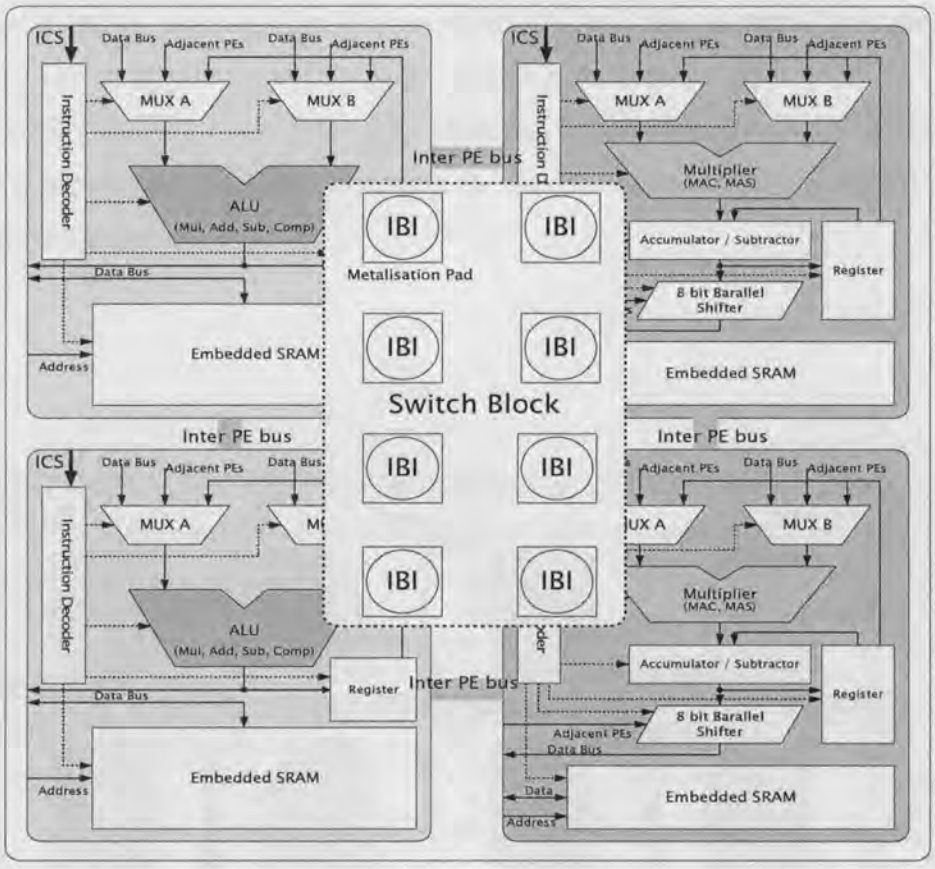


Figure 3.7: Quad-PE

3.7 UnitCAP Chip Architecture

The CAP chip consists of 4 sets of UnitCAP. Each UnitCAP has an array of 16 heterogeneous S-PEs and PA-PEs. Figure 3.8 shows the UnitCAP chip architecture. The configurable interconnectivity is realised through the input multiplexer in each PE. The detailed description of interconnection between the PEs will be described in Chapter 6.

3.8 Conclusions

The heterogeneous types of PE architecture for communication and multimedia signal processing have been described. The adoption of the PE architecture can accelerate the

performance where intensive bit-level computation and digital signal processing is required and achieve more flexibility compare with homogeneous types of PE array. The suggested PE architecture has been fully modelled and its functionality verified using SystemC at high-level. The details regarding the system level modelling of the PE will be introduced in Chapter 7.

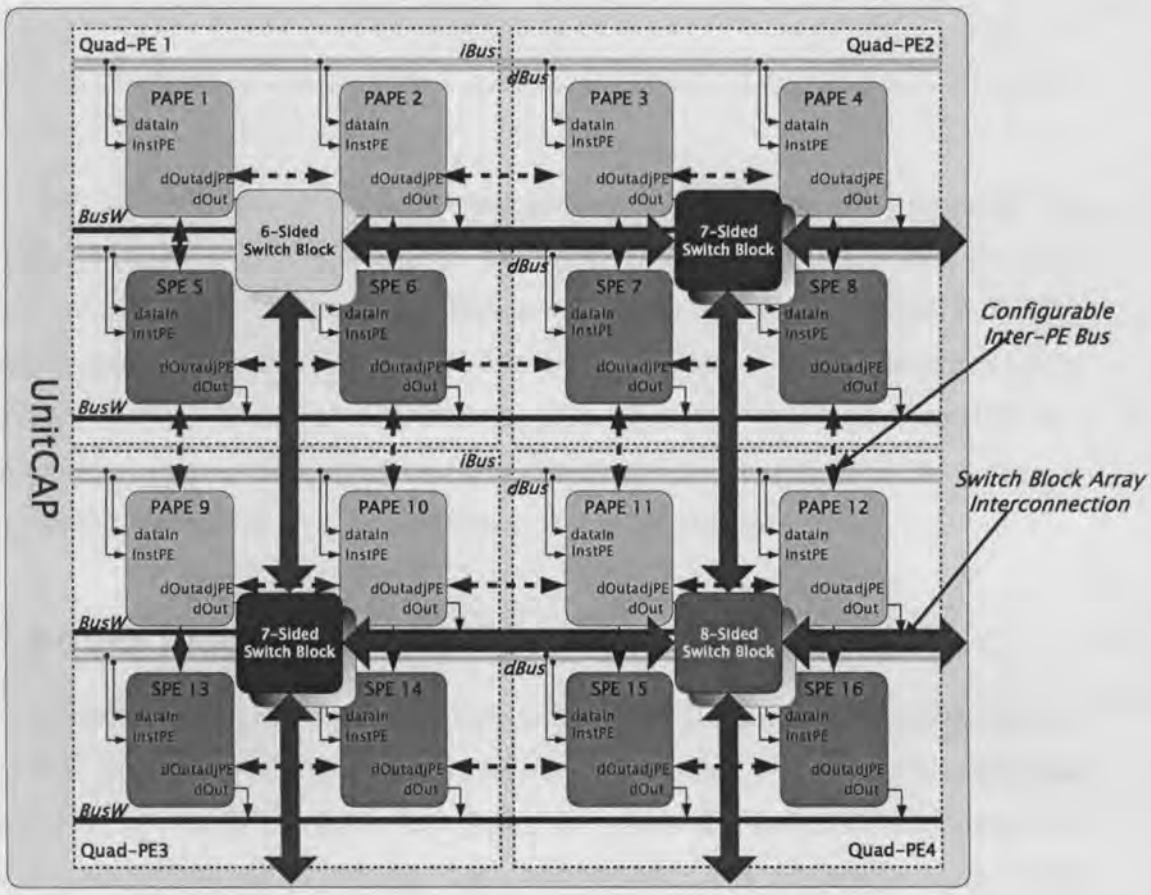


Figure 3.8: UnitCAP Chip Architecture

Chapter 4
Architecture of ICS Chip

The ICS chip comprises the Switch Blocks, ICS RISC, program memory, data memory, data frame buffers and DMA controller. The ICS chip is a control processor which controls the CAP chip via the IBIA as well as the overall system. The ICS_RISC provides control and address signals and data to the system as a whole. The switch blocks configure each PE based on the current program instruction. The high bandwidth data Interface Unit enables efficient transmission of data and instructions within the system. In this chapter, the detailed architecture of the ICS chip is described.

4.1 Switch Block

The Switch Block provides data from/to each PE and also provides instruction data to each PE. Three types of Switch Block, 6-sided, 7-sided and 8-sided provide optimized interconnection within the ICS chip. Figure 4.1 shows the Switch Block architecture which connects between the PEs and other Switch Blocks. The architecture of the Switch Block is similar to conventional Switch Blocks in Field Programmable Gate Arrays (FPGA) [32]. The lines in the figure represent switches to connect data/instruction data within the PEs, Switch Blocks and the ICS chip. A pass transistor design is used to optimize performance and minimise area, allowing a completely free configuration for each PE.

4.2 ICS_RISC

The ICS_RISC is a 32-bit dedicated RISC control processor. The ICS_RISC controls the execution of the PE array and provides control and address signals to program/data memory, the data frame buffers and the DMA controller. It has a 3 stage pipelined architecture that is Fetch (F), Decode (D) and Execute (E). To cope with the iterative nature of DSP arithmetic, it has 16 sets of loop buffers so as to provide direct instruction to instruction decoding instead of fetching from program memory in each case. This significantly reduces bus utilization allowing for improved performance and lower power dissipation. Moreover 32 general purpose registers and specialized addressing modes are provided for optimized communication and multimedia signal processing. For detailed architecture descriptions refer to Appendix B.

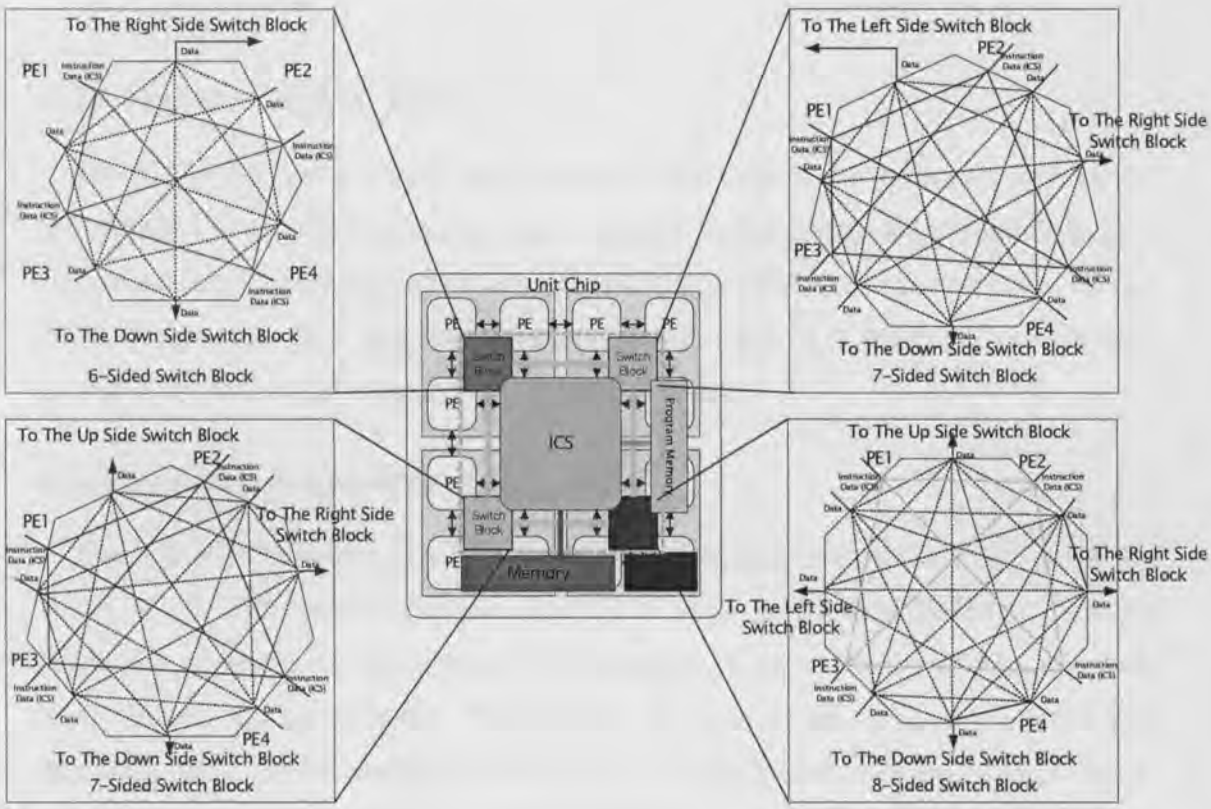


Figure 4.1: Architecture of Switch Block: A 6-sided Switch Block, 7-sided Switch Block and 8-sided Switch Block

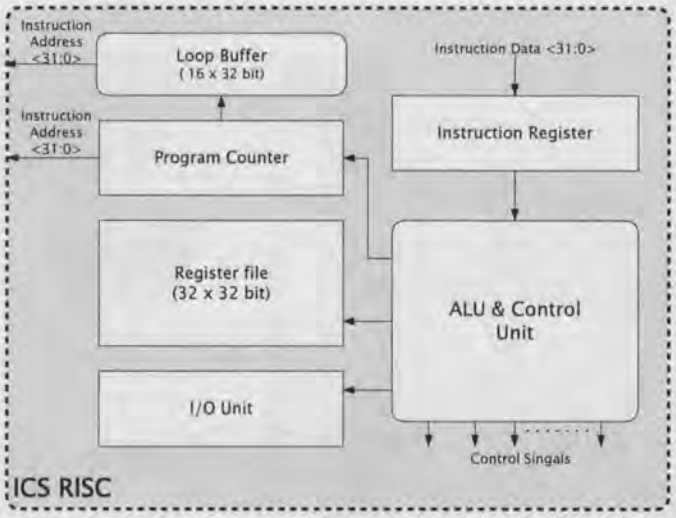


Figure 4.2: Architecture of ICS_RISC 32-bit dedicated Control Processor

4.2.1 Features of ICS_RISC

The ICS_RISC has a simple and efficient architecture. It has a harvard architecture and simple 3 stage pipelined architecture. Memory access during the execution stage is carried out using load/store instructions only and all operations, except load/store, PE and DMA operations, are register-to-register within the ICS_RISC. This provides improvements in the performance and power dissipation.

4.2.2 System Components of ICS_RISC

The ICS_RISC consists of a 32 × 32-bit general purpose register, a program counter which is the 32th general purpose register, a 16 × 32-bit loop buffer to generate instruction addresses for iterative sets of instructions, a status register (N:Negative/Less than, Z:Zero, C:Carry/Borrow, V:Overflow), an instruction register for instruction decoding, ALU, shifter, multiplier and 32-bit data input/output registers [30,31]. Figure 4.3 shows the detailed architecture of the ICS_RISC.

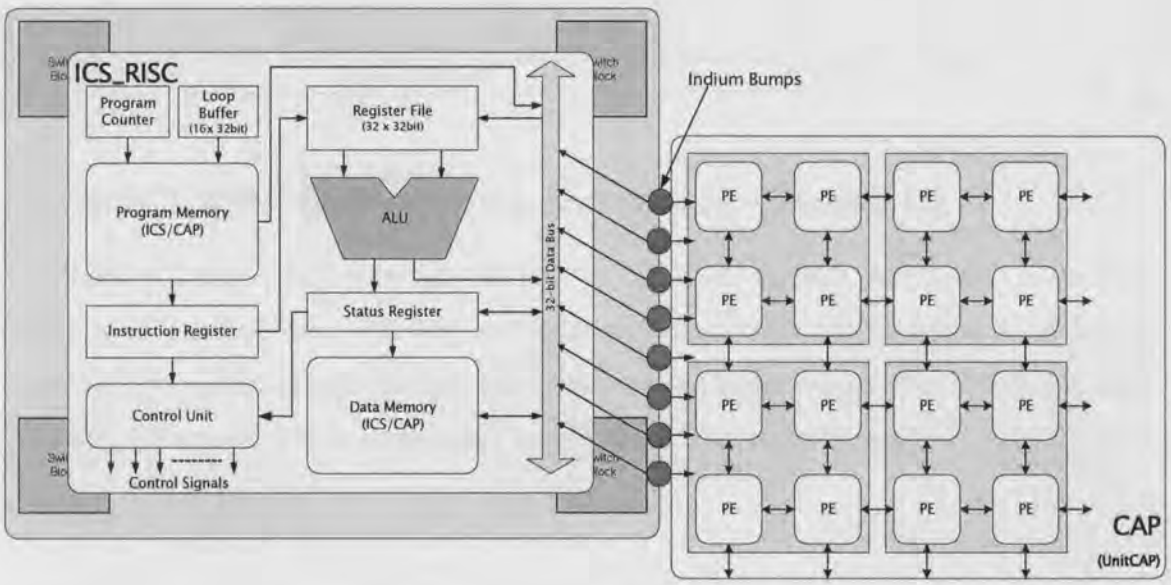


Figure 4.3: A detailed architecture of the ICS_RISC

4.2.3. Types of Instruction Set

Table 4.1 describes the instruction set and instruction processing components of the 3D-SoftChip. All control instructions are executed in the ICS chip, while computation instructions, such as arithmetic and logical operations for PEs are executed in the CAP chip using various computation methods (SISD, SIMD, MISD, MIMD). The detailed instruction set is described in Appendix A.

Table 4.1: Types of Instruction Set

Function	Processing Component
Move	ICS
Arithmetic (S-PE, PA-PE)	CAP
Logical (S-PE)	CAP
Arithmetic	ICS
Logical	ICS
Branch	ICS
Load	ICS
Store	ICS
Addressing Mode/Loop Buffer Addressing	ICS
PE Control	ICS

PE Configuration	ICS
Program/Data Load (ICS,PE Program/ Data for PE)	ICS
DMA Control	ICS

4.2.4. ICS_RISC Instruction Set Architecture – Version 1.0

Table 4.2 shows the instruction set architecture (ISA) for ICS_RISC. This is the first version of the ISA, more efficient and dedicated instructions can be added is needed. It has 50 instructions, largely divided into arithmetic and logic, branch, data transfer, bit and bit-test, PE control, DMA control and lastly a loop buffer instruction.

Table4.2: Instruction Set Summary (ICS_RISC ISA Version1)

Mnemonic	Operation	Operands	Flags
ARITHMETIC AND LOGIC INSTRUCTIONS			
ADD	Add Two Registers	Rd, Rs1, Rs2	N,Z,C,V
ADDI	Add Register and Constant	Rd, Rs1, #I	N,Z,C,V
SUB	Subtract Two Registers	Rd, Rs1, Rs2	N,Z,C,V
SUBI	Subtract Register and Constant	Rd, Rs1, #I	N,Z,C,V
MUL	Multiply Two Registers	Rd, Rs1, Rs2	N,Z,C,V
MULI	Multiply Register and Constant	Rd, Rs1, #I	N,Z,C,V
AND	Logical AND Registers	Rd, Rs1, Rs2	N,Z,C,V
ANDI	Logical AND Register and Constant	Rd, Rs1, #I	N,Z,C,V
OR	Logical OR Registers	Rd, Rs1, Rs2	N,Z,C,V
ORI	Logical OR Register and Constant	Rd, Rs1, #I	N,Z,C,V
XOR	Logical XOR Registers	Rd, Rs1, Rs2	N,Z,C,V
XORI	Logical XOR Register and Constant	Rd, Rs1, #I	N,Z,C,V
NOT	Logical NOT Registers	Rd, Rs1, Rs2	N,Z,C,V
NOTI	Logical NOT Register and Constant	Rd, Rs1, #I	N,Z,C,V
BRANCH INSTRUCTIONS			
BREQ	Branch if Equal (Z=1)	PC, Offset	None
BRNE	Branch if NOT Equal (Z=0)	PC, Offset	None
JMP	Unconditional Branch (PC=PC+Offset)	PC, Offset	None
CMP	Compare Registers	Rs1, Rs2	N,Z,C,V
CMPI	Compare Register and Constant	Rd, #I	N,Z,C,V
DATA TRANSFER INSTRUCTIONS			
MOVA	Move between Registers (Rd=Rs1)	Rd, Rs1	None
MOVAI	Move between Reg. & Const. (Rd=Const)	Rd, #I	None
MOVB	Move between Registers (Rd=Rs2)	Rd, Rs2	None
MOVBI	Move between Reg & Const. (Rd=Const)	Rd, #I	None
MSR	Move Register to Status Register(SR=Rs1)	SR, Rs1	None

MSRI	Move Imm value to Status Register(SR=#I)	SR, #I	None
MRS	Move Status Register to Register(Rs1=SR)	Rs1, SR	None
LD	Load indirect with Register (Rd=Mem[Rb])	Rd, Rb	None
ST	Store indirect with Register (Mem[Rb]=Rd)	Rd, Rb	None
BIT AND BIT-TEST INSTRUCTIONS			
LSL	Logical Shift Left	Rd, Rs1	N,Z,C,V
LSR	Logical Shift Right	Rd, Rs1	N,Z,C,V
ASR	Arithmetic Shift Right	Rd, Rs1	N,Z,C,V
ROT	Rotate	Rd, Rs1	N,Z,C,V
PE CONTROL INSTRUCTIONS			
PECON4	PE Word-Length Configuration (4-bit)	None	None
PECON8	PE Word-Length Configuration (8-bit)	None	None
PECON16	PE Word-Length Configuration(16-bit)	None	None
PECON32	PE Word-Length Configuration (32-bit)	None	None
PESEL	Select certain PE (PE0~PE15)	None	None
PEMODH	PE Operation mode (Horizontal mode)	None	None
PEMODV	PE Operation mode (Vertical mode)	None	None
PEMODC	PE Operation mode (Circular mode)	None	None
PEEXEH	Execute specific program to each PEs in the same Horizontal line	None	None
PEEXEV	Execute specific program to each PEs in the same Vertical line	None	None
PEEXEC	Execute specific program to each PEs in the same Circular line	None	None
DMA INSTRUCTIONS			
LDPEPRG	Load Program Data from Program memory to Instruction decoder in PE.	addrMem	None
LDDFB	Load large amount of processing data for PEs from Data Memory to Data Frame Buffer	addrMem, addrDFB	None
LDPEDATA	Load large amount of processing data for PEs from DFB to Embedded SRAM in PE	addrDFB, addrSRAM	None
WBREG	Write back processed data in Embedded SRAM to the registers in the ICS_RISC	addrSRAM	None
WBDFB	Write back processed data in Embedded SRAM to DFB	addrSRAM, addrDFB	None
STDFB	Load large amount of processed data in PEs from Data Frame Buffer to Data Memory	addrDFB, addrMem,	None
LOOP BUFFER INSTRUCTION			
LBEN	Generate an Iterative Set of Instruction Addresses (16 sets of Loop Buffer)	PC	None

4.3 High Bandwidth Data Interface Unit

The high bandwidth data interface unit allows the efficient transfer of data within the 3D-SoftChip. Two sets of data frame buffer and the DMA controller make it easy to transfer large amounts of data. Multiple sets of program memory support run-time program switching and, because of this dynamic reconfigurable feature, adaptive computing is possible. The data memory has a variable word width so it can easily be combined to build wider/deeper memories and thus increase flexibility for different application programs. The DMA instructions and data flow for the DMA controller can be seen in Figure 4.4. A detailed description of the operations of the DMA instructions can be seen in Appendix A.

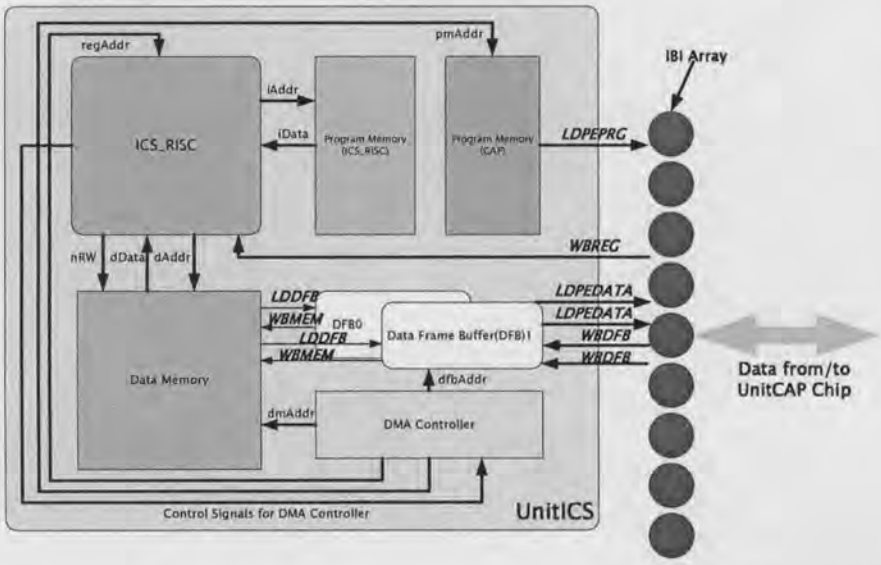


Figure 4.4: DMA Controller Architecture and Instructions for DMA Controller

4.4 Conclusions

The ICS chip architecture has been described in this chapter. The system components in the ICS Chip allow it to efficiently supply data and instructions to the PEs through the IBIA. The PE array can be freely configured due to the highly controllable characteristic of the switch block. This allows more than sufficient adaptability/flexibility for adaptive

computing systems. Moreover, the DMA controller enables transfer of the bulk data fast and effectively through the 3D-SoftChip.

Chapter 5
Architecture of UnitChip

The 3D-SoftChip consists of 4 sets of UnitChip. Each UnitChip has one UnitCAP and one UnitICS. As described in the chapter 3, the UnitCAP comprises 16 sets of heterogeneous arrays of S-PEs and PA-PEs and the UnitICS consists of a switch block, a 32-bit dedicated RISC control processor, a high bandwidth data interface unit, 2 sets of data frame buffers and program/data memory for both the ICS and the PE array. In this chapter, the UnitChip architecture and its pipeline operation mechanism which can maximize the computational throughputs [3], are described.

5.1 UnitChip Architecture

As mention above, the UnitChip is a combination of the UnitCAP and the UnitICS chip and four UnitChips form the complete 3D-SoftChip. Figure 5.1 illustrates the overall architecture of the UnitChip. The control, data and instructions transfer through the IBIA to the UnitCAP, and the processed data from the UnitCAP can be rapidly transferred back to the ICS_RISC to be manipulated and stored in the data memory.

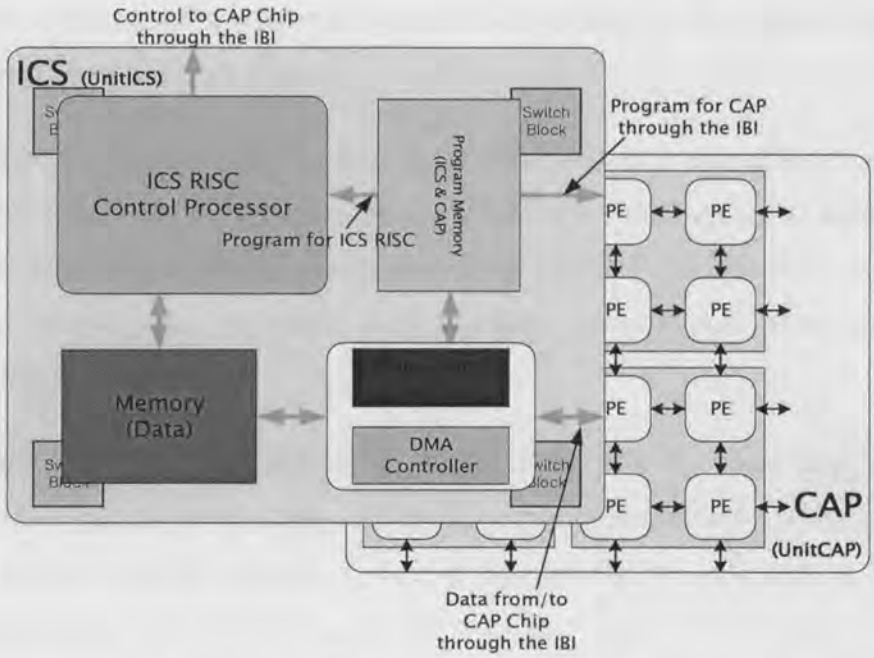


Figure 5.1: Overall Architecture of the UnitChip.

5.2 Pipelined Operation Mechanism of UnitChip

Table5.1: Pipelined UnitChip Operation Mechanism

	Stage (1)	Stage (2)	Stage (3)	Stage (4)	Stage (5)	Stage (6)
ICS_RISC Instructions	LDPEPRG PESEL, PEMODH,V,C	LDDFB	LDDFB PESEL PEMODH,V,C	LDPEPRG PECON4,8,16,32 PEEXEH,V,C	PECON4,8,16,32 PEEXEH,V,C WBREG,WBDFB	PECON4,8,16,32 PEEXEH,V,C WBMEM
PEs Op.				Execute (1-1)	Execute (1-2)	Execute (2-1)
PROGRAM for PEs (in Local memory)	Load PRGM for PEs (1)		PRGM for PEs(1-1)	Load PRGM for PEs (2)	PRGM for PEs(2-1)	PRGM for PEs(2-2)
Data Frame Buffer 0		Load Data for PEs (1)	Data for PEs (1-1)	Data for PEs (1-2)	Write back Execution (1-1) results	
Data Frame Buffer 1			Load Data for PEs (2)		Data for PEs(2-1)	Data for PEs(2-2)
Memory						Write back Execution(1-1) results

* Dark-sided boxes: DMA Control Instructions

Table 5.1 illustrates the pipelined operation mechanism of UnitChip to improve its performance. The detailed explanation is as follows.

- **STEP 1 - LOAD PROGRAM FOR PEs:** The first operation is to load 16 instruction words for PEs from program memory to the instruction decoder in the each PE, the row and column decoder in the UnitCAP can specify a certain PE to load the programs, depending on the desired computational mode (e.g, SIMD, MIMD).

- **STEP 2 - LOAD PROCESSING DATA FOR PEs (1):** Load large amount of processing data for PEs from data memory to data frame buffer. The start address of memory and an amount of data to transfer can be indicated by the DMA instructions.

- **STEP 3 - LOAD PROCESSING DATA FOR PEs (2):** Load the processing data for PEs from data frame buffer to embedded SRAM in each PEs. The row and column decoder in the UnitCAP can specify a certain PE to load the processing data.

- **STEP 4 - EXECUTE PEs:** Execute the PE array

- **STEP 5 - RELOAD PROGRAM FOR PEs (1):** Reload 16 instruction words from program memory to the instruction decoder in each PE, the row and column decoder in the UnitCAP can again specify a certain PE to load the programs to.

- **STEP 6 - RELOAD PROCESSING DATA FOR PEs (2):** Reload the processing data for PEs from data frame buffer to each PEs, the row and column decoder in the UnitCAP can specify a certain PE to load the data into.

- **STEP 7 - WRITE BACK PROCESSED DATA TO DFB:** Write back processed data from embedded SRAM in each PEs to data frame buffer

- **STEP 8 - TRANSFER PROCESSED DATA TO Memory:** Transfer large amount of processed data from data frame buffer to memory

5.3 Area Estimations and Constraints

Table 5.2 shows the feasible estimated area of 3D-SoftChip components. The performance of the integrated circuits largely depends on integration density. The tight area constraints can be achieved through more integration density, which means it can maximize benefit from large scaled integration. The area constraints should be tight in order to achieve the best performance.

Table 5.2: Area Estimation and Constraint of UnitChip (Target Technology: 0.13 um Process)

Component	Estimated Area
S-PE	60 um × 60 um
PA-PE	60 um × 60 um
IBIA	15 um × 15 um
One Quad-PE	130 um × 130 um
UnitCAP	500 um × 500 um
CAP(4×4 UnitCAP)	1100 um × 1100 um
CAP(16×16 UnitCAP)	2200 um × 2200 um
ICS_RISC	300 um × 300 um

5.4 Conclusions

As explained above, by using the pipeline operation mechanism that is a 6-stages pipelined architecture, the performance of the UnitChip can be 6 times more improved. This pipelined operation is another distinguished character to accelerate the computational throughput as the computation is executed simultaneously as much as the pipelined stages.

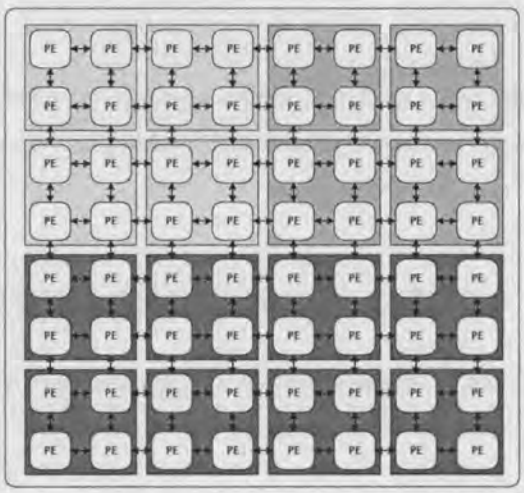
Chapter 6
Interconnection Network

In this chapter, the three hierarchical interconnection architectures: Inter-PE bus, Switch Block Array interconnection and IBIA, will be introduced along with the configurable nature of the Inter-PE bus using the input operand multiplexer in each of the PEs.

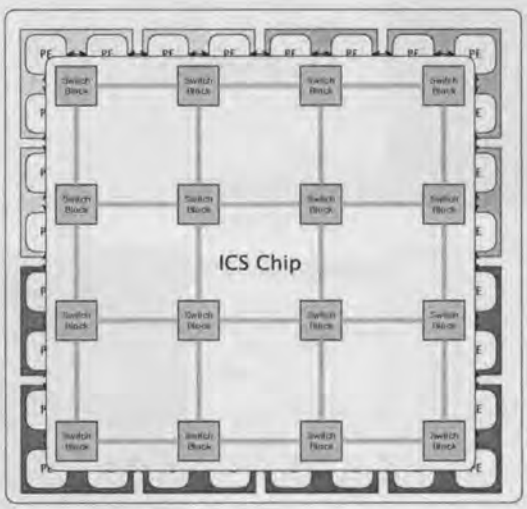
6.1 Hierarchical Interconnection Architecture

The interconnection network of the 3D-SoftChip can be broken into three hierarchical levels. The Inter-PE bus between PEs in the CAP chip is the first level. This local interconnection network has a 2D-mesh architecture providing nearest-neighbor interconnection between the PEs. The second level of the interconnection network is the switch block array interconnection. This supports longer interconnections on the ICS chip but also has a basic 2D-mesh architecture. The last hierarchical level of interconnection is the IBIA. With progression of technology to ever decreasing semiconductor geometry scales, the prediction of interconnection delay and the portion of interconnection delay in the total system delay are crucial factors. It is also a major factor in the limitation of overall system performance. To overcome these problems, 3D interconnection technology using Indium bump becomes very attractive because it supports a very high bandwidth coupled with a very low inductance/capacitance (and thus low power dissipation) and can be readily utilized to achieve an interconnect array with a pitch as

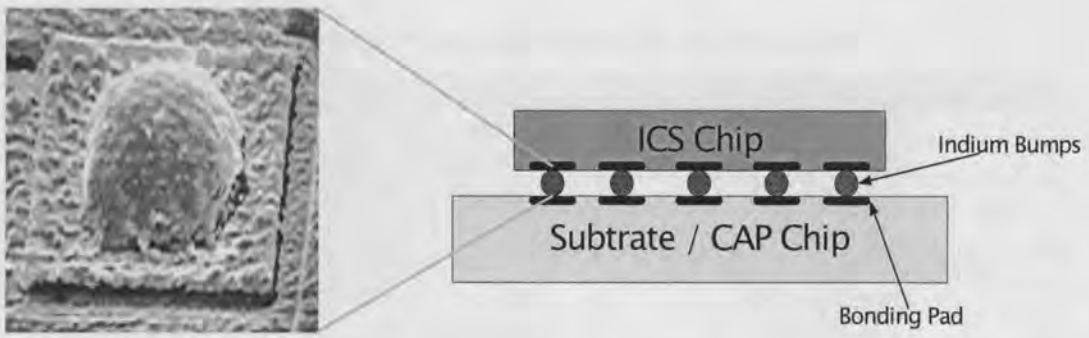
low as 10μm. The development of 3D integrated systems will allow improvements in packaging costs, performance, reliability and a reduction in the size of the chips [15]. However, any other equivalent 3D interconnection technology could also be applied to realize this interconnection level within the 3D-SoftChip architecture. Figure 6.1 shows the three hierarchical interconnection networks.



(a) PE Array Interconnection Network: 2D-mesh interconnection for local interconnection



(b) Switch Block Array Interconnection Network: 2D-mesh interconnection for long interconnection



(c) Indium Bump Interconnection: Single indium bump after reflow

Figure 6.1: Three hierarchical Interconnection Networks

6.1.1. PE and Switch Block Array Interconnection

6.1.1.1 Programmable Nature of PE Array Interconnection

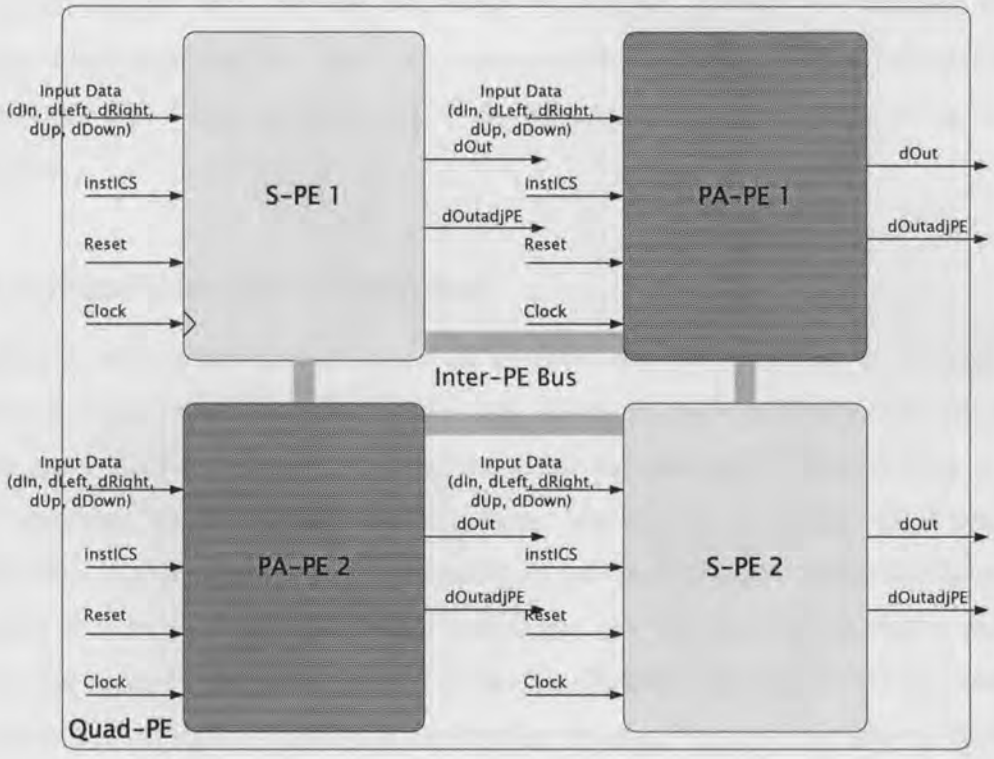


Figure 6.2: Quad-PE and Programmable Interconnect Architecture

Table6.1: Inter-PE Bus (IPB) interconnection connectivity

IPB Signal Name	Source(Output)	Destination(Input)
IPB1	SPE1(dOutadjPE)	PAPE1(dLeft)
IPB2	SPE1(dOutadjPE)	PAPE2(dUp)
IPB3	PAPE1(dOutadjPE)	SPE1(dRight)
IPB4	PAPE1(dOutadjPE)	SPE2(dUp)
IPB5	PAPE1(dOutadjPE)	Next Quad-PE(SPE1(dLeft))
IPB6	PAPE2(dOutadjPE)	SPE1(dDown)
IPB7	PAPE2(dOutadjPE)	SPE2(dLeft)
IPB8	PAPE2(dOutadjPE)	Downside Quad-PE(SPE1(dUp))
IPB9	SPE2(dOutadjPE)	PAPE1(dDown)
IPB10	SPE2(dOutadjPE)	PAPE2(dRight)
IPB11	SPE2(dOutadjPE)	Next Quad-PE(PAPE2(dLeft))
IPB12	SPE2(dOutadjPE)	Downside Quad-PE(PAPE1(dUp))

Figure 6.2 shows the Quad-PE architecture and Inter-PE interconnection architecture [3]. Because of the input multiplexer in each PE, the connectivity can be readily configured. The input multiplexer can choose certain input operands from among the 6 different inputs; data input, data from left side, right side, upward side and down side PE (dIn, dLeft, dRight, dUp, dDown) and each PE's output (dOutadjPE) becomes input operand to the neighbour PEs. Table 6.1 describes the connectivity within one Quad-PE and indicates that it can be configured by the PE programming according to the target application.

6.1.2. Indium Bump Interconnection

Indium is an excellent material to use as an interconnect material due to its excellent adhesion to most metals, including aluminum, which is the metallization for the pads used in most VLSI technologies. Indium has a low melting point, which implies a low work hardening coefficient, allowing for direct bonding on processed VLSI wafers. Additionally, it provides excellent mechanical as well as electrical connectivity (contact resistance < 1 mΩ per bump). Reflow techniques can be used for flexibility and to increase the bump height to width ratio as needed. Such techniques can also be used to incorporate self-alignment features to the bonding process. Figure 6.3 illustrates 3D filp-chip wafer bonding technology using indium bump interconnection arrays.

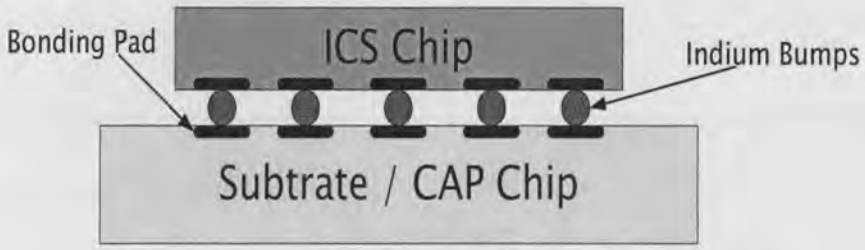


Figure 6.3: 3D Flip-Chip wafer bonding technology using Indium Bump Interconnection Arrays

6.2 Conclusions

The three hierarchical interconnection network architectures have been described. With the exception of the 3D interconnection there are similar to conventional interconnection architectures in reconfigurable systems. The Inter-PE bus provides configurable connectivity with 2D mesh architecture and the switch block interconnection offers longer interconnection in the ICS chip. Lastly, the IBIA presents vertical interconnection between the two separated chips providing a high bandwidth, high speed, low power memory bus, reducing and eliminating the needs for external memory resources.

Chapter 7
High-level modeling of 3D-SoftChip using SystemC

In this chapter, the high-level modelling of 3D-SoftChip using SystemC will be introduced. Firstly, an overview of SystemC, Computer Aided Design (CAD) environment for SystemC will be briefly described, followed by a presentation of the high-level simulation output waveforms for each of the 3D-SoftChip components and analysis of these. Finally, some conclusions are provided.

7.1 SystemC Overview

SystemC is a C++ class library and design methodology which can effectively design a software algorithm, hardware architecture, interface with SoC and system level designs. System-level modelling, quick simulation to validate and optimize design and HW architecture and various software algorithms explorations can all be achieved using conventional C++ development environments. The current system design methodology is for the system engineer to write high-level language (C, C++, Matlab etc.) programs to verify the concepts and algorithms at system-level. After the concepts and algorithms are validated, the high-level modelled designs are manually converted to the Hardware Description Languages (VHDL, Verilog-HDL) in order to implement the hardware. But

this approach gives rise to a number of problems, such as errors arising from the manual conversion from C to HDL, a disconnection between the system level model and HDL model and conversion limitation as design sizes is get ever bigger and more complex. As a result of this, new C language based system design languages are starting to emerge as a new design methodology. Figure 7.1 shows the conventional system design in contrast to a SystemC based design methodology.

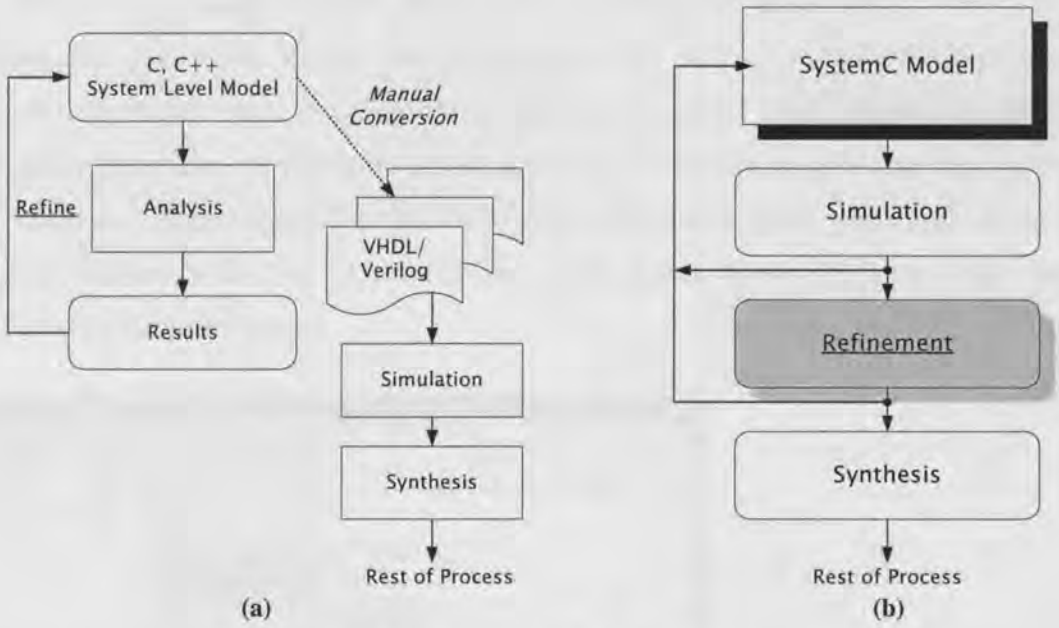


Figure 7.1: System Design Methodology:
(a) Conventional Design Methodology, (b) SystmC Design Methodology (*Source: www.systemc.org)

The system design methodology using SystemC has many advantages over the conventional system design methodology including increased more productivity and reliability from the progressive refinement process and the use of a single language. In the design methodology using SystemC, the time consuming manual conversion process is no longer necessary because the high-level modelled code becomes a more reliable and high performance hardware model while hardware concepts and timing constructs can be added through the progressive refinement process. More productivity can be achieved by using a single design language, the high-level modelled SystemC code can result in smaller code that is easer to write as well as relatively faster simulation time, moreover

the testbench code for functional verification at high-level can be reused at any level or design stage[19,20].

7.1.1. CAD Environment for SystemC

As described above, SystemC is a C++ class library, which means any conventional C++ compiler can be a CAD development environment for SystemC. Any Unix, linux or PC based C++ compiler can be used, however, in this research, the PC based CAD environment (Microsoft Visual C++ Version 6.0) has used to compile the high-level modelled SystemC code because of its easy accessibility. Once SystemC code is compiled, the results are stored in various types of file. The most common file type for the results is a Value Change Dump (VCD) type and the GTKWave waveform viewer is used to validate VCD type of results. The figure below shows the Visual C++ and GTKWave waveform viewer.

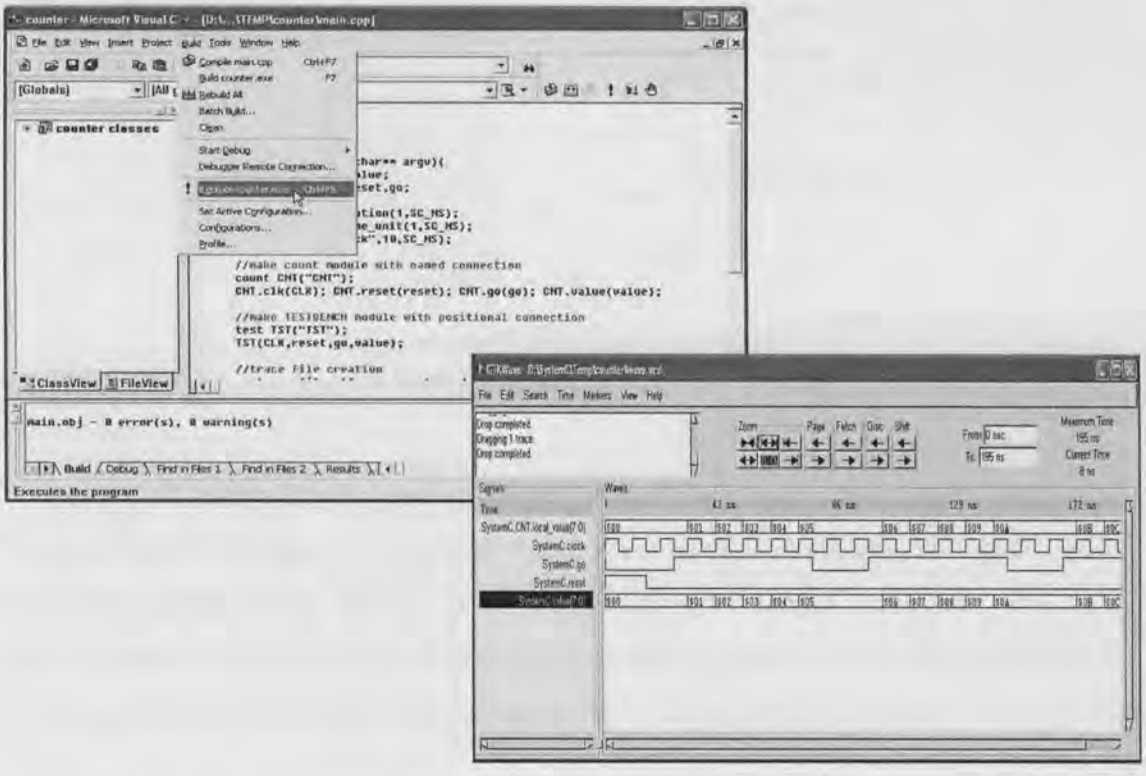


Figure 7.2: The CAD Environment for SystemC; Visual C++ Version 6.0, GTKWave Waveform Viewer.

7.2 System-level Modeling of 3D-SoftChip

In this section, the high-level modelled single Standard-PE, Processing-Accelerator-PE, ICS_RISC and UnitChip will be introduced with output simulation waveform. The functionality of these components has been fully verified. For a more detailed description of the system-level modelling of 3D-SoftChip see Appendix B.

7.2.1. Standard-PE

The detailed architecture of the S-PE was introduced in Chapter 3. Based on the architecture, it has been high-level modelled using SystemC. Figure 7.3 shows the block diagram and SystemC file structure of the S-PE.

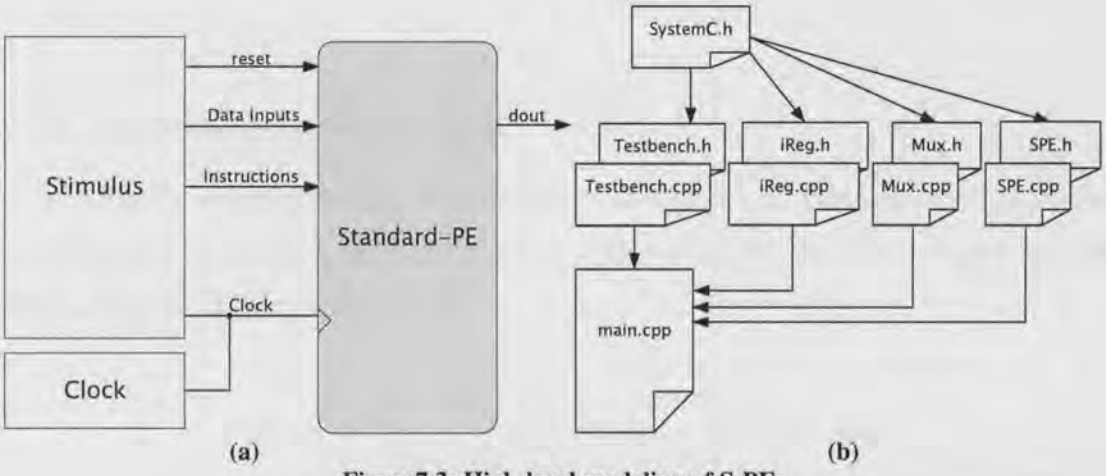


Figure7.3: High-level modeling of S-PE:
(a) S-PE block diagram, (b) file structure of S-PE

Figure 7.4 shows the output waveform of the S-PE execution results after ALU instructions between data from internal registers and embedded SRAM. The input signals (dIn, dLeft, dRight, dUp, dDown) have been selected by the input multiplexer. The ALU output signals can be seen in the dOut, and dOutadjPE signals. The functionality of the S-PE was confirmed by checking the output result.

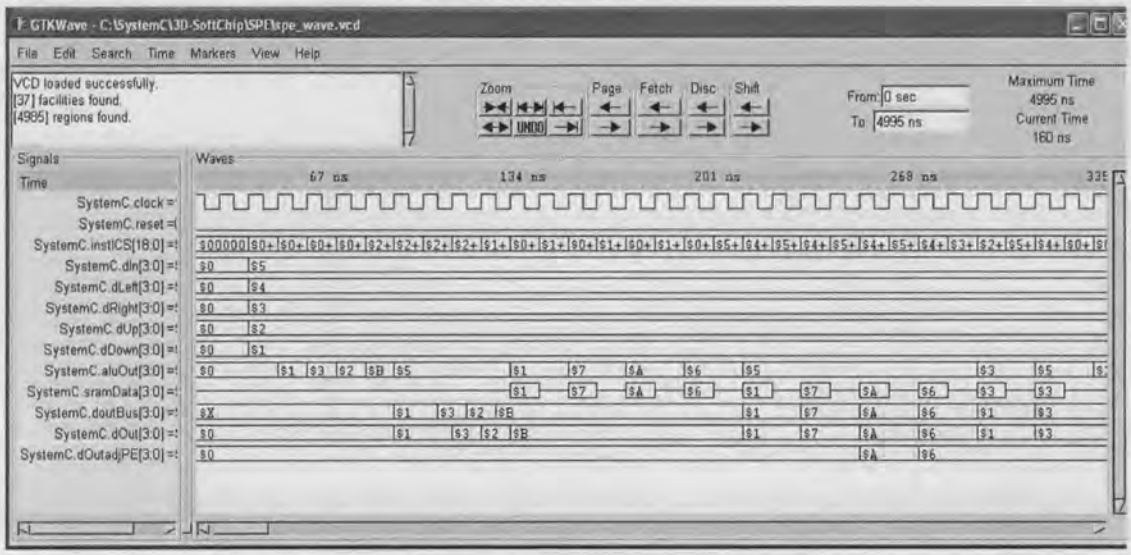


Figure 7.4: The Output Waveform of S-PE

7.2.2. Processing Accelerator-PE

The PA-PE architecture has been described in Chapter 3. The high-level modelling was executed from this description. Figure 7.5 shows the PA-PE block diagram and file structure for the SystemC modelling.

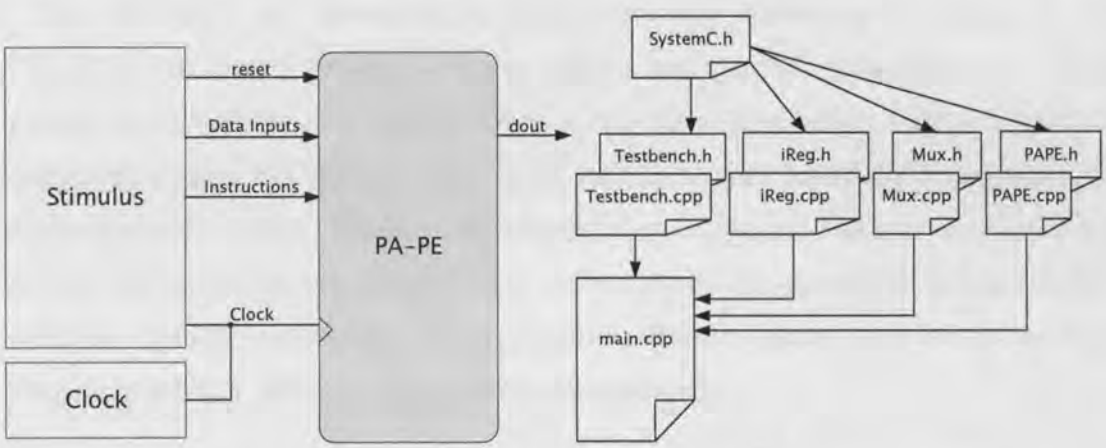


Figure 7.5: High-level modeling of PA-PE:
(a) PA-PE block diagram, (b) file structure of PA-PE

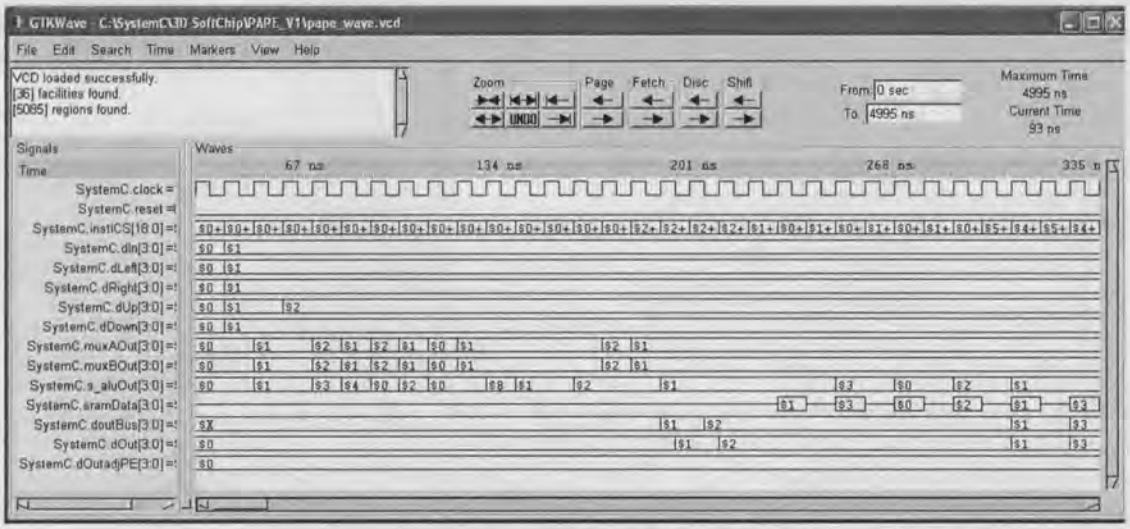
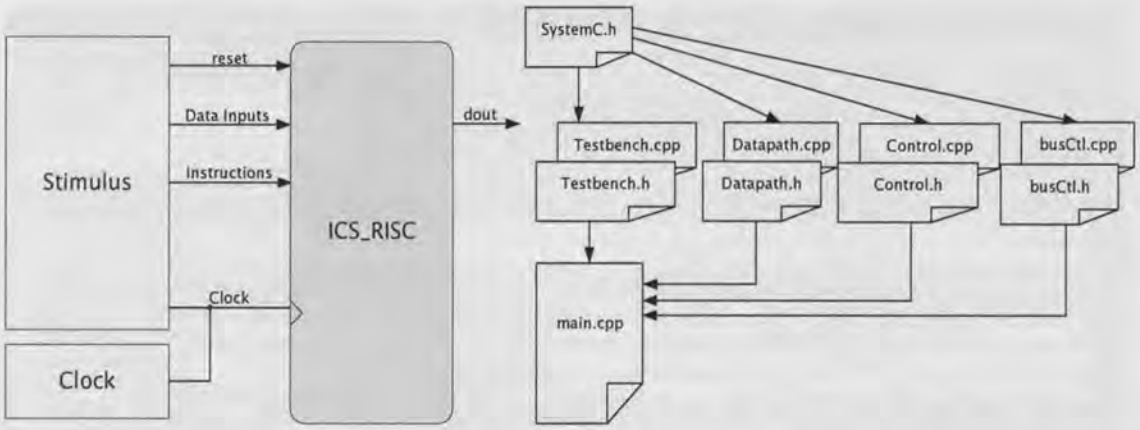


Figure 7.6: The Output Waveform of PA-PE

The figure above shows the output waveform of the high-level modelled PA-PE. The selected input operand through the input multiplexer executes the ALU instruction (MAC, MAS, Shift, etc) and the results are then stored to the embedded SRAM. The output signal shows the operation executed as required.

7.2.3. ICS_RISC

The ICS_RISC and instruction set architecture was introduced in Chapter 4. The ICS_RISC can largely be classified into control and datapath units. The 32 × 32-bit general purpose register, a program counter, a 16 × 32-bit loop buffer, a status register, an instruction register, ALU, shifter, multiplier and 32-bit data input/output registers form the datapath architecture. The fetch, decoding and execution unit make up the control unit. Additionally, a bus control unit is used to control the 32-bit operand A, operand B, data write bus, input bus and output bus to avoid data collision. Figure 7.7 shows the top block diagram of the ICS_RISC and its SystemC file structures.



(a) **(b)**

Figure 7.7: High-level modeling of ICS_RISC:
(a) ICS_RISC block diagram, (b) file structure of ICS_RISC

The output waveform shows the results after execution of simple loop and ALU instructions. Figure 7.8 shows the pseudo code for the instructions. The circle in figure 7.9 which is written as a loop instruction indicates the internal general purpose register address. It increases as programmed and the other circle presents the output result of the ALU operations.

```
//Simple Loop & ALU Instruction

MOV   R0,    #0;     //Simple Loop Inst
MOV   R1,    #1;
MOV   R2,    #2;
MOV   R3,    #3;
MOV   R4,    #4;
MOV   R5,    #5;
MOV   R6,    #6;
MOV   R7,    #7;
MOV   R8,    R0;
MOV   R9,    R1;
MOV   R10,   R2;
MOV   R11,   R3;
MOV   R12,   R4;
MOV   R13,   R5;
MOV   R14,   R6;
MOV   R15,   R7;     //End of Loop Inst.
AND   R16,   R8 , R9;        //ALU Inst
OR    R17,   R10, R11;
XOR   R18,   R12, R13;
ADD   R19,   R14, R15;
SUB   R20,   R14, R15;       //End
```

Figure 7.8: The Psuedo Code for ICS_RISC

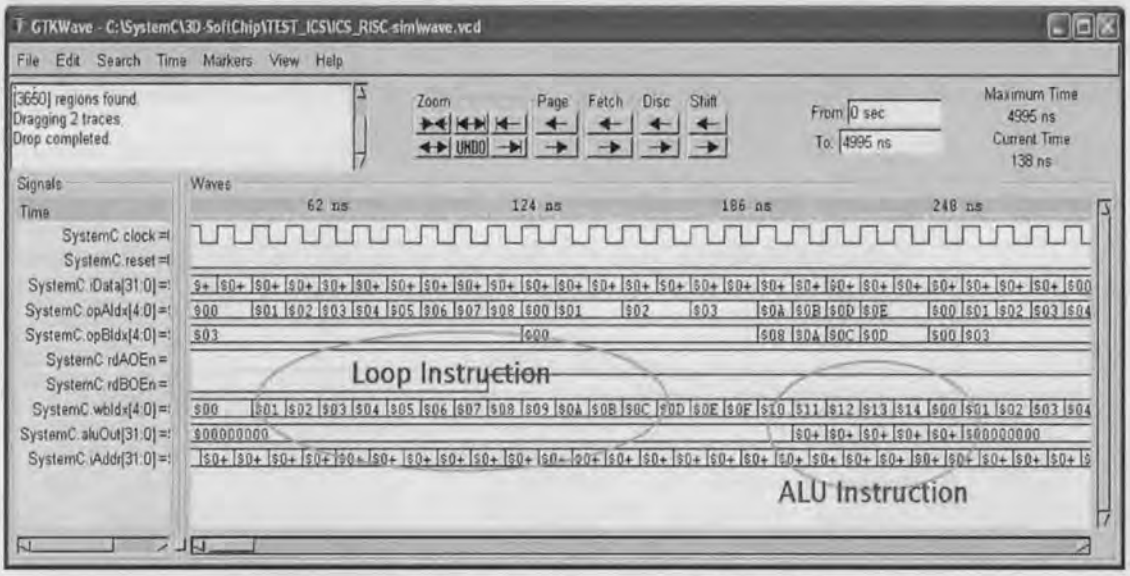


Figure 7.9: The Output Waveform of ICS_RISC

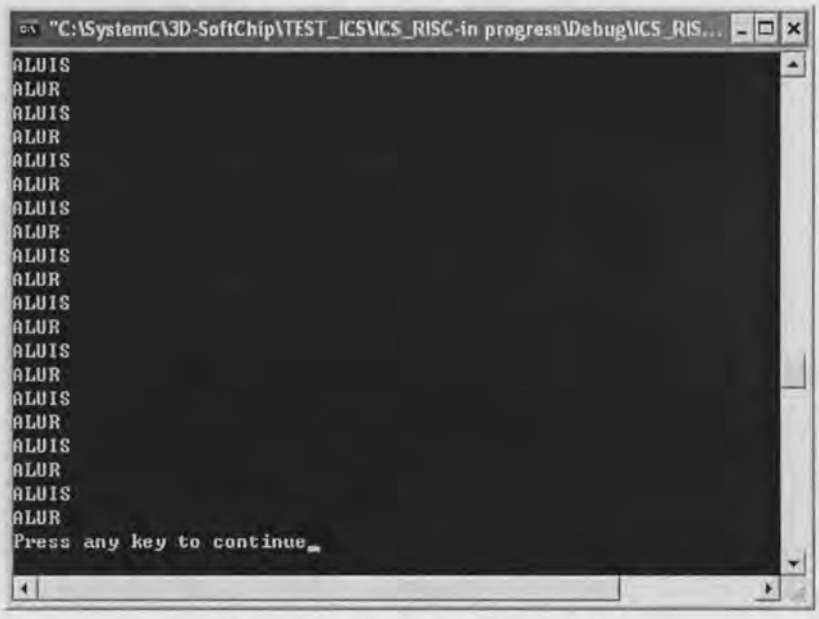


Figure 7.10: The Instruction Index

Figure 7.10 shows the instruction index during ICS_RISC instruction execution for debugging purposes. The instruction index was perfectly matched with the instruction of the pseudo code.

7.2.4. UnitChip

The composition of the UnitCAP and UnitICS becomes the UnitChip. It can be largely divided into 4 kinds of sub-SystemC files, that is ICS_RISC, Memory, DMA and UnitCAP. As described in Chapter 5, the architecture and the pipelined operation mechanism can be identified in the high-level system simulation results.

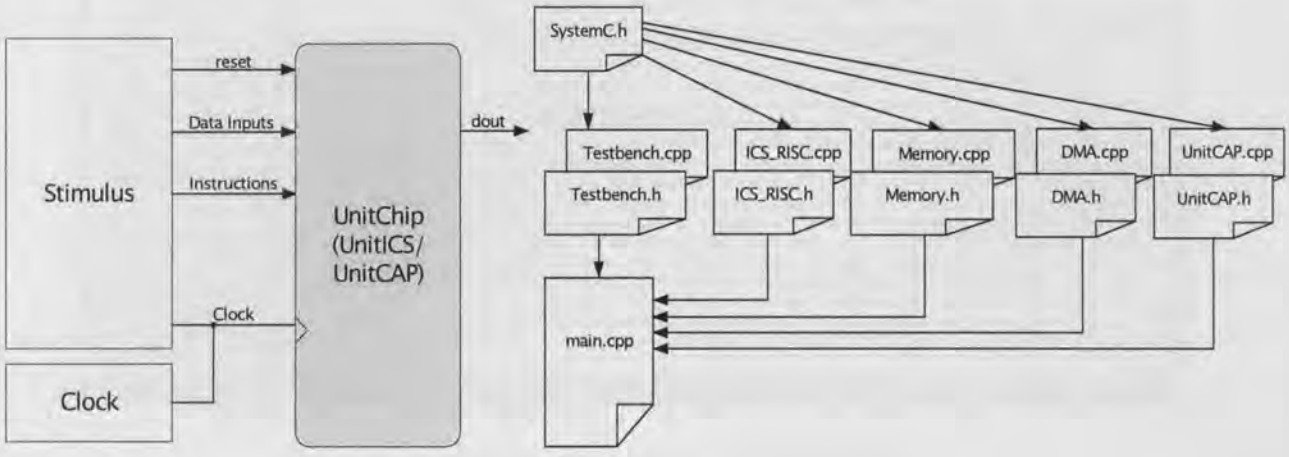


Figure 7.11: High-level modeling of UnitChip:
(a) UnitChip block diagram, (b) file structure of UnitChip

Figure 7.11 illustrates the UnitChip block diagram and SystemC file structure of the UnitChip. Each sub-SystemC block's functionality has been described before, the UnitChip is a simple combination (port-mapping) of these sub-blocks at the top module. The simple ALU instruction has been mapped in this high-level modelled UnitChip. The simulation result shows its functionality. In figure 7.12, the upper side circle indicates the ICS_RISC operation introduced before, and lower circle shows the PEs operations, which is the execution of simple ALU functions for the PEs with parallelism. The signal named as a PE1.dOut means the output signal from PE1. The functionality can be verified by checking these signals (from PE1~PE16) and is as expected.

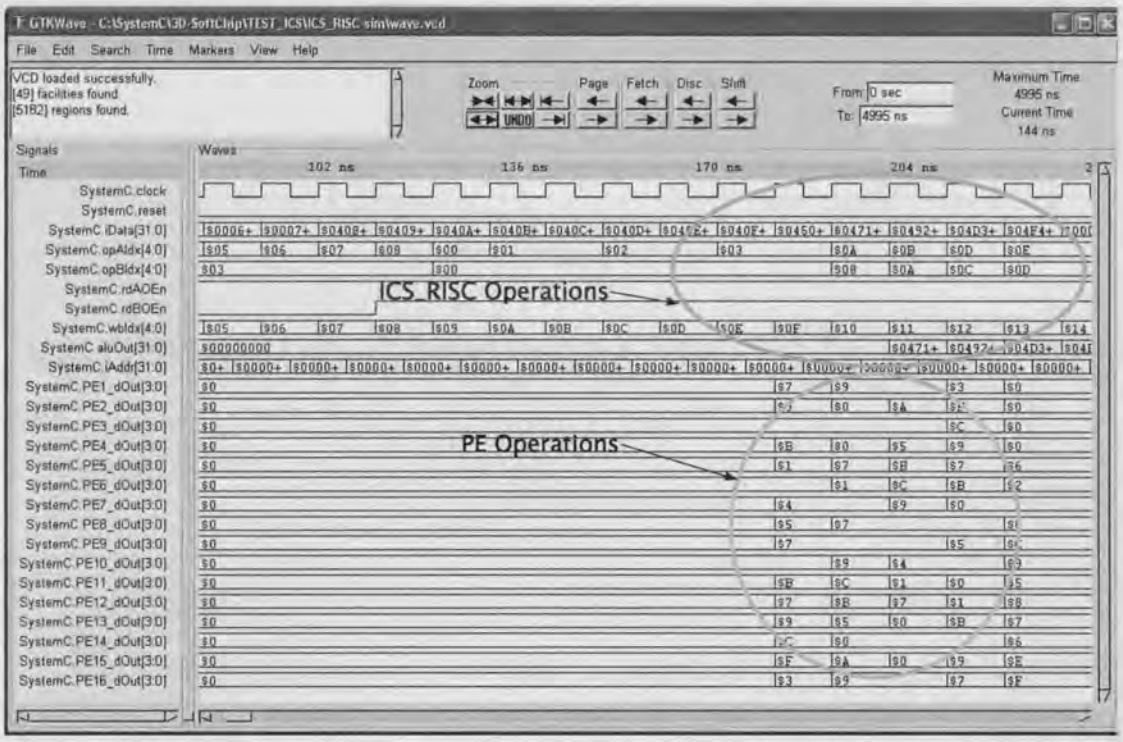


Figure 7.12: The Output Waveform of UnitChip

7.3 Conclusions

In this chapter, the overview of SystemC and its CAD tool develop environment has been introduced. The high-level system modelling and functional verification of the 3D-SoftChip using SystemC has been described and some simulation results provided. The waveforms show the correct functionality for each of the sub-blocks and for the top module of the UnitChip.

Chapter 8

Application Mapping for 3D-SoftChip

The MPEG4 Full Search Block Matching Motion Estimation Algorithm (FBMA) has been applied to the high-level system modeled 3D-SoftChip to verify its functionality and demonstrate its architectural superiority. The hand-crafted assembler code for implementation of the algorithm becomes the input stimulus of the system-level modeled 3D-SoftChip. The performance will be analyzed in compraison with a conventional DSP processor, Application Specific ICs (ASICs) and MorphoSys.

8.1 Full Search Block Matching Algorithm (FBMA)

Motion estimation (ME) is introduced to exploit the temporal redundancy of video sequences and is an indispensable part of video compression standards such as the ISO/IEC, MPEG-1, MPEG-2, MPEG-4 and the CCITT, H.261/ITU-T, H.263 etc. Since ME is computationally the most demanding portion of the video encoder, it can take up to 80% of total computation time and it can be a major limiting factor for the performance. Among the many different ME algorithms, FBMA is one of the most widely used in hardware, despite its high computational cost because it has the optimal performance and lowest control overhead. The block matching motion estimation algorithm compares a specific sized block of pixels in the current frame with a range of equally sized pixel blocks in the previous frame to find the best match (minimum difference) between two of the blocks. The position of the best matched block can then be encoded as a motion

- **STEP 1 – LOAD REF. BLOCK DATA INTO PE ARRAY SRAM**: The first operation is to load reference block data ($I_k(m,n)$) into embedded SRAM in each PE in the array.

- **STEP 2 – EACH PE MOVES THIS DATA TO INTERNAL REGISTER:** Each PE moves the reference data from the embedded SRAM into an internal register so it is available to be used for calculation of SAD values for the entire search window.

- **STEP 3 – LOAD FIRST SEARCH POSITION BLOCK DATA INTO PE ARRAY SRAM:** The block data for the first search position ($I_{k-1}(m+dx,n+dy)$) is then loaded into the embedded SRAM in each PE in the array ready for calculation of the SAD value between the reference block and this first search position.

- **STEP 4 – EACH PE EXECUTES SUBTRACTION AND ABSOLUTE VALUE COMPUTATION:** In this step, each PE carries out a subtraction operation between the reference block data and the current search position in SRAM, the absolute value of this resulting difference is stored as the absolute difference value for that block position.

- **STEP 5 – PARTIAL SUMMATION (1):** In this step every odd columned PE performs a partial sum operation of its absolute difference value with the value from the PE to its immediate right in the array, the result is stored as a double-word value across both PEs.

- **STEP 6 – PARTIAL SUMMATION (2):** In this step the two partial sums computed in the previous step are summed in the same way, every odd columned PE pair sums its result with the result from the PE pair to its right, this result is stored as a quad-word value across all four PEs in each row.

- **STEP 7 – PARTIAL SUMMATION (3):** In this step the column wise operation carried out in step 5 is repeated row wise to accumulate another set of partial sums,

in this case, however, the second row of PEs accumulated its result with the result from the row above, while the third row of PEs accumulates its result with the result from the row below.

- **STEP 8 – PARTIAL SUMMATION (4):** In this final partial sum accumulation, the second row of PEs sums its result with the result from the third row, producing the total SAD value for that search position.

- **STEP 9 – WRITE BACK RESULT DATA TO THE ICS_RISC:** Finally the resultant SAD value calculated in STEP 8 is written back to the internal register in the ICS_RISC for comparison with the previous minimum and updating of the motion vector if applicable.

- **STEP 10 – REPEAT STEPS 4 TO 9:** The next search position data block can be loaded into the SRAM in the PE array while the SAD calculation is being carried out for the current search position so once the result had been written back the calculation of the SAD for the next search position can be begun immediately.

8.3 Performance Analysis

Figure 8.3 shows the performance comparison of the 3D-SoftChip with a DSP processor, several ASICs and MorphoSys for matching on 8×8 reference block against its search area of 8 pixels displacement. There are 81 candidate blocks (27 iterations) in each search area [33]. In the 3D-SoftChip, as described above, the number of processing cycles for one candidate block is just 7 clock cycles (each UnitChip computes one quarter block, so with 4 UnitChips one complete block is computed every 7 cycles), so the total number of processing cycles for the 3D-SoftChip becomes 567 (81 iterations of 7 cycles each).

The number of clock cycles required is very close to that reported for MorphoSys, with just 4 UnitChips, this, however, can readily be improved simply by increasing the number of UnitChips on a scaled up 3D-SoftChip. A 4×4 UnitChip array, for example, would have an effective throughput of one block every 142 cycles. In addition to this, considering the characteristics of the 3D system, there are other significant advantages.

Data dependency is largely eliminated so there after the initial set-up there is a 100% PE utilisation. The reference and candidate block data can be moved into the embedded SRAM in the PE concurrently with array execution, so the PEs can operate continuously. Also low power consumption can be achieved through a minimisation of the number of data accesses, because most of data manipulation can be executed within the PE array. Most importantly, however, because all memory is directly accessible within the 3D-SoftChip via the IBIA there are effectively zero external data reads and thus power consumption will be greatly improved over all the other approaches.

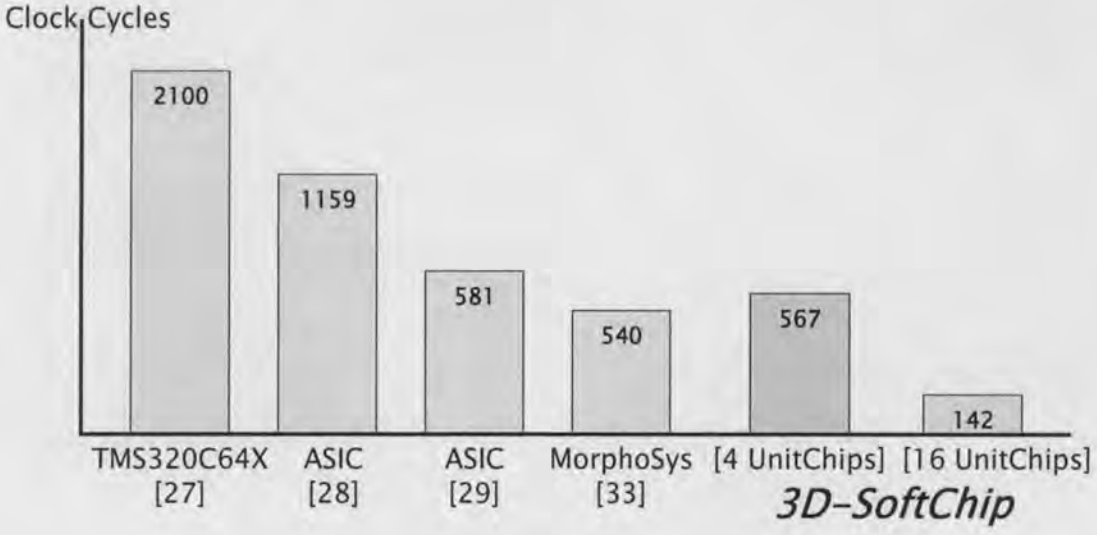


Figure 8.3: Performance comparison for Motion Estimation

When comparing with the performance of the DSP processor and dedicated ASICs, the performance of the suggested 4×4 UnitChip 3D-SoftChip has remarkable advances with a theoretical capability of more than 3.8 times the performance. Given its wide applicability/adaptability to any number of other applications, the performance achieved compared to these dedicated processors is a potentially enormous advancement. This clearly demonstrates the architectural superiority of the suggested novel 3D-SoftChip.

8.4 Conclusions

In this chapter, the mapping of the implemented MPEG4 full search block matching algorithm has been applied to the system-level modelled 3D-SoftChip in order to demonstrate its architectural superiority. According to the described results, the proposed 3D-SoftChip architecture has the potential for a more than 3.8 times performance improvement over conventional systems. The suggested 3D-ACSoC is clearly a highly suitable system for the coming giga-scaled integrated computing age.

Chapter 9
Conclusions

In this chapter, the contribution of this thesis will be summarised and future research work will be suggested.

9.1 Contributions

In this thesis, a novel 3D vertically integrated adaptive system-on-chip architecture as a next generation computing system along with its functional verification and the mapping of an MPEG4 motion estimation algorithm has been presented. The suggested architecture has a number of advantages compared with conventional current generation reconfigurable/adaptive computing systems, such as wide applicability, various and powerful computation methods, adaptive word-length configuration and benefits from the architecture including 3D interconnect performance, reliability and a reduction in the size of the chips (and thus the cost), as described before. As outlined in chapter 5.3, the size of total chip as described is relatively small at around 1.1 mm^2 for an array of 2×2 UnitChips or 2.2 mm^2 for an array of 4×4 UnitChips. This is based on a 4-bit word-length for the PEs so there is also ready potential to extend to a wider word-length (8-bit word-length) and more integration of the PEs to maximise the computational throughput and benefits from large integration. Moreover, the ICS_RISC can also be readily extended on the upper chip layer by adopting advanced computation algorithms and dedicated instructions for specific applications to allow more efficient controllability and

performance over the current relatively simple ICS_RISC design. As minimum feature sizes continue to decrease in more advanced chip fabrication processes the inherent scalability of the UnitChip design means that the array size can simply be increased to within the constraints of the maximum die size to realise ever more power adaptive computing systems.

The performance of the execution of the MPEG4 full search block matching motion estimation algorithm has been shown to be more 3.8 times improved over current generation processors. Due to these significant performance, power and cost advantages it can be shown that the suggested 3D-ACSoC is one of the most suitable architectures for the next generation of computing system.

Moreover, the suggested advanced HW/SW co-design and verification methodology can accelerate the reliability and significantly reduce the design time, especially the time and effort required for verification. This thesis indicates a highly promising research direction for future adaptive computing systems and an advanced and efficient HW/SW development methodology for ever more complicated SoCs.

9.2 Future Work

As introduced in the suggested design methodology, the high-level modelling and functional verification has been carried out, the next task is the architectural explorations to obtain an optimized HW specification. The method to explore various architecture options is through parameterized memory, data frame buffer and DMA controller modelling using SystemC, followed by simulation with various HW configurations so as to find the best HW specification. The use of the parameterized modelling method makes the architecture exploration considerably easier, the parameter values can simply be changed in the SystemC code. Figure 9.1 shows the SystemC modelling of the parameterized memory. Once the optimum HW specification is decided, the rest of the procedure can be executed with any conventional hardware design method, such as full and semi-custom design and the SW design should be concurrently performed so that the novel concept of an adaptive system-on-chip computing system can be realised.

```
//Paramteterized RAM

#ifdef RAMT_H
#define RAMT_H
#include "systemc.h"

template <class T, int size = 100>

SC_MODULE(ram) {
        sc_in<bool>   clock;
        sc_in<bool>   nRW;   //Read/Write
        sc_in<int>    addr;  //Address
        sc_inout<T>   data;  //Parameterized Word-length

void ram_proc();
SC_HAS_PROCESS(ram);

ram(sc_module_name name_, bool debug_ = false);
        sc_module(name_), debug(debug_)
{
 SC_THREAD(ram_proc);
sensitive << clock.pos();
buffer = new T[size];

if (debug) {
        cout << "Running constructor of" << name() << endl;
        cout << "Number of location is" << size << endl;
}
}
private :
        T* buffer;
        const bool debug;
};

template <class T, int_size>
void ram<T, size>::ram_proc()
{
while(true) {
        wait();
        if (nRW) {
                data = buffer[addr];
        } else {
                buffer[addr] = data;
        }
}
}
}
#endif
```

Figure 9.1: The parameterized Memory modeling example using SystemC

Bibliography

[1] E. Mirsky and A. DeHon, "MATRIX : A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," Proc. IEEE Symp. FPGAs for Custom Computing Machines, pp.157-166, April 1996

[2] S.C. Goldstine, H. Schmit, B.Mihai, S. Cadambi, M.Matt, R.R.Taylor, "PipeRench : A Reconfigurable Architecture and Compiler," IEEE Computer, pp.70-77, April 2000

[3] S. Hartej, L. Ming-hua, L. Guangming, J.K. Fadi, B. Nadar, M.C.F Eliseu, "MorphoSys : An Integrated reconfigurable system for data-parallel and computation-intensive applications", IEEE transactions on computers, Vol49, No.5, pp.465-481, May 2000

[4] E. Waingold et al. "Bring it all to software : RAW Machines", Computer, Vol30, Issue9, pp.86-93, Sept.1997

[5] T. Miyamori , K. Olukotun, "REMARC : Reconfigurable Multimedia Array Coprocessor", Proc. ACM/SIGDA FPGA98, Monterey, Feb.1998

[6] C. Ebeling et al., "Architecture design of reconfigurable pipelined datapaths", Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on, pp.23-40, 21-24. Mar.1999

[7] Goldstein, S.C, et al, "PipeRench: a reconfigurable architecture and compiler", Computer, Vol.33, Issue4, pp.70-77, April 2000

[8] D.Chen, J. Rabaey, "PADDI: Programmable arithmetic devices for digital signal processing", VLSI Signal Processing IV, IEEE Press, 1990

[9] Elixent Ltd, "The Reconfigurable Algorithm Processor", http://www.elixent.com/products/white_papers.htm

[10] Triscend Corp., "Triscend A7S Configurable System-on-Chip Platform", http://www.triscend.com/

[11] Motorola Inc., "MRC6011: Reconfigurable compute Fabric(RCF) Device", http://www.motorola.com/semiconductors/

[12] Nick Tredennick, Brion Shimamoto, "Special Report: Do-it-all devices", IEEE Spectrum, pp.37-40, Dec. 2003.

[13] L. Guangming, "Modeling, Implementation and Scalability of the MorphoSys Dynamically Reconfigurable Computing Architecture," PhD thesis, Univ. of California, Irvine, 2000

[14] S. Eshraghian, S. Lachowicz, K. Eshraghian, "3-D Vertically Integrated Configurable Soft-Chip with Terabit Computational Bandwidth for Image and Data Processing", Proc. MIXDES'2003, Lodz, Poland, June 26-28, 2003

[15] A. Rassau, G. Alagoda, A. Ehrdardt, S. Lachowicz, K. Eshraghian, "Design Methodology for a 3D-SoftChip Video Processing Architecture", 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI2002), Orlando, Florida, U.S.A, pp.324-329, July 14-17, 2002

[16] QuickSilver Technology Inc., "Adapt2400 ACM architecture Overview", http://www.quicksilvertech.com/pdfs/Adapt2400_Whitepaper_0404.pdf

[17] picoChip Design Limited, "PC102 Product Brief", http://www.picochip.com

[18] S. Eshraghian, "Implementation of Arithmetic Primitives Using Truly Deep Submicron Technology (TDST)", Ms thesis, Edith Cowan University, 2004

[19] Open SystemC Initiative, "The Functional Specification for SystemC 2.0", http://www.systemc.org/

[20] Open SystemC Initiative, "SystemC 2.0.1 Language Reference Manual Rev 1.0", http://www.systemc.org/

[21] International Technology Roadmap for Semiconductors (ITRS), "International Technology Roadmap for Semiconductors 2003 Edition", http://public.itrs.net/

[22] AMDREL Consortium, "Existing Functional Level Reconfigurable Implementation Platforms", http://vlsi.ee.duth.gr/amdrel/deliverables.html

[23] IZM, "3D System Integration", http://www.pb.izm.fhg.de/izm/015_Programms/010_R/

[24] Joyner J.W., Zarkesh-Ha P.J, Meindl J.D, "Global Interconnect Design in a Three-Dimensional System-on-a-Chip", IEEE Transactions on VLSI systems, Vol 12, Issue4, pp.367-372, April 2004

[25] J.W. Joyner, et al., "Impact of three-dimensional architectures on interconnects in gigascale integration", IEEE Trans. VLSI Syst. Vol. 9, pp.922-928, Dec. 2001

[26] Kaustv. Banerjee, et al, "3-D ICs: A Novel Chip Design for Improving Deep-Submicrometer Interconnection", Proceedings IEEE Special Issues on Interconnections, Vol. 89, No 5, pp602-633, May 2001

[27] Texas Instruments, "TMS320C6000 Assembly Benchmarks", http://www.ti.com/sc/docs/products/dsp/c6000/benchmarks/67x.htm

[28] K M Yang, M-T Sun and L.Wu, "A Family of VLSI Design for Motion Compensation Block Matching Algorithm", IEEE Trans. on Circuits and Systems, Vol. 36, No 10, pp1317-25, October, 1989

[29] C. Hsieh and T.Kin, "VLSI Architecture for Block-Matching Motion Estimation Algorithm", IEEE Trans. on Circuits and Systems for Video Technology, Vol.2, pp169-175, June 1992

[30] Yeong-don Bae, "Basic Microprocessor Design", http://www.donny.co.kr

[31] Yap Zi He, "Building A RISC Microcontroller in an FPGA, "http://www.opencores.org/projects/riscmcu/

[32] Michael Shyu, Guang-Ming Wu, Yu-Dong Chang and Yao-Wen Chang, "Generic Universal Switch Blocks", IEEE Trans. on Computer, Vol. 49, Issue 4, pp348-359, April 2000

[33] Hartej Singh, "Reconfigurable Architectures for Multimedia and Data-Parallel Application Domains", PhD Thesis, University of California Irvine, 2000

[34] R.Gao, D.Xu and J.P.Bently, "Reconfigurable Hardware Implementation of an Improved Parallel Architecture for MPEG-4 Motion Estimation in Mobile Applications", IEEE Trans. on Consumer Electronics, Vol. 49, No4, pp1383-1390, November 2003

[35] J.Rabay et al, "Reconfigurable Computing: The Solution to Low Power Programmable DSP", Proc. ICASSP 97, Munich, Germany, April 1997

[36] Stylianos Perissakis et al, "Embedded DRAM for a Reconfigurable Array", Proc. of VLSI99, June 1999

[37] Chameleon Systems Inc. "CS2000TM Reconfigurable Communications Processor", CS2000TM Advanced Product Information, http://www.chameleonsystems.com

1 ICS_RISC Instruction Set Architecture Version 1.0

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immedi.(1word)	0	0	0	0	0	0	0	Opcode				Rd					Immediate(4,8,16-bit)															
Immedi.(2word)	0	0	0	0	0	0	1	Opcode				Rd					Unused															
	Immediate (32-bit)																															
Register	0	0	0	0	0	1	0	Opcode				Rd					Rs2					Rs1					Unused					
LB Addressing	0	0	0	0	0	1	1	Opcode				Rd					Rs2					Rs1				RWEn	Unused				
Shift / Rotate	0	0	1	0	1	0	0	Shift			x	Rd					ShiftAmt					Rs1					Unused					
Load	0	0	1	0	1	0	1	0	x			Rd					x					Rb					Unused					
Store	0	0	1	0	1	1	0	1	x			Rd					x					Rb					Unused					
Branch	0	0	1	0	1	1	1	Cond			x	Offset																				
PE Control	0	1	0	1	0	0	0	PE Op			x	Opmode			Config		PE Sel										Unused					
DMA Control	1	DMA Op			DFB Sel	Amount of Data to Transfer						Start address of DFB (Sou/Dst)					SRAM/ RegSel	Start address of SRAM/ICS Reg(S/D)					Mem Sel	Start address of Program/Data Memory (Sou/Dst.)								
Multiply	0	1	1	1	1	1	1	x				Rd					Rs2					Rs1					Unused					
Dedicated Instructions	**Not yet decided**																															

Opcodes				Mnemonics	Description (Immediate)	Description (Register)
0	0	0	0	**MOVA**	Rd = Immediate	Rd = Rs1
0	0	0	1	**MOVB**	Rd = Immediate	Rd = Rs2
0	0	1	0	**AND**	Rd = Rd & Immediate	Rd = Rs1 & Rs2
0	0	1	1	**OR**	Rd = Rd | Immediate	Rd = Rs1 | Rs2
0	1	0	0	**XOR**	Rd = Rd ^ Immediate	Rd = Rs1 ^ Rs2
0	1	0	1	**NOT**	Rd = ~ Immediate	Rd = ~Rs1
0	1	1	0	**ADD**	Rd = Rs1 + Immediate	Rd = Rs1 + Rs2
0	1	1	1	**SUB**	Rd = Rs1 – Immediate	Rd = Rs1 – Rs2
1	0	0	0	**CMP**	Compare Rs1 and Immediate	Compare Rs1 and Rs2
1	0	0	1	**MSR**	Status Register = Immediate	Status Register = Rs1
1	0	1	0	**MRS**	N/A	Rs1 = Status Register

Shift			Mnemonics	Description
0	0	0	**LSL**	Shift Left
0	0	1	**LSR**	Shift Right
0	1	0	**ASR**	Arithmetic Shift Right
1	0	0	**ROT**	Rotate

Cond			Mnemonics	Description
0	0	0	**EQ**	Equal
0	0	1	**NE**	Not Equal
0	1	0	**AL**	Always (Unconditional)

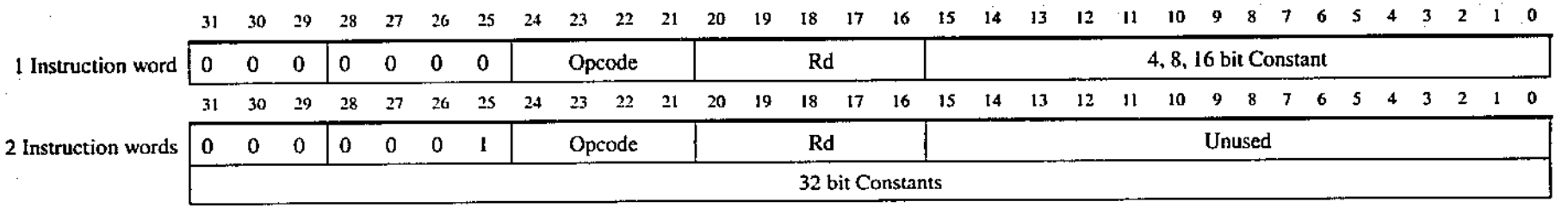
PE Operations			Mnemonics	Description
0	0	0	**PECONF**	Configuration of each PEs (4,8,16,32 bits)
0	0	1	**PESEL**	To select certain PE (PE0 ~ PE15)
0	1	0	**PEMODE**	To select PE operation modes (Horizontal/Vertical/Circular modes)
0	1	1	**PEVEXE**	To execute specific program to each PEs in the same vertical line
1	0	0	**PEHEXE**	To execute specific program to each PEs in the same horizontal line
1	0	1	**PECEXE**	To execute specific program to each PEs in the same circular line

DMA Operations			Mnemonics	Description
0	0	0	**LDPEPRG**	Load maximum16 program data from Program memory to Embedded SRAM in PEs
0	0	1	**LDDFB**	Load large amount of processing data for PEs from Memory to Data Frame Buffer
0	1	0	**LDPEDATA**	Load large amount of processing data for PEs from DFB to Embedded SRAM in PE
0	1	1	**WBREG**	Write back processed data in Embedded SRAM to the Registers in the ICS_RISC
1	0	0	**WBDFB**	Write back processed data in Embedded SRAM to DFB
1	0	1	**WBMEM**	Write back processed data in DFB to Data Memory

1.1 Instruction descriptions

- Immediate addressing : Short immediate values : 4,8,16 bit (1 instruction word), Long immediate value : 32 bits (2 instruction words)

  Rd = Rd op Immediate (4,8,16,32 bit)



  Description: The processed data from PEs and immediate data from regFile or data memory can be manipulated in the ICS_RISC so it can process 4,8,16,32bit data.

- Register addressing :

  Rd = Rs1 op Rs2

  Description: Rs1 and Rs2 indicates the address of internal regFile(32 sets of 32bit data(32 × 32bit)). The opcode identifies the operations and the manipulated data between Rs1 and Rs2 is stored in the register which indicated by Rd.

- LB Addressing:

  Rd = Rs1 op Rs2

  Description: When the LB Addressing becomes active, the sources of addresses become a Loop Buffer. It has 16 depths of looping capacity.

- Shift / Rotate :

  Rd = Rs1 Shift by Amount

  Description: According to the shiftCtl and shiftAmt, the shifter can shift the input operands.

- Load :

  Rd = Mem [Rb]

Description: Rd in the regFile can load the data from data memory address which indicates by Rb.

- Store :

  Mem [Rb] = Rd

  Description: Data in the regFile can store to the data memory address which indicates by Rb

- Branch :

  If (Cond) PC = PC + Offset

  Description: According to the Cond signals, the Program Counter value can increase as much as offset value.

- Multiply :

  Rd = Rs1 * Rs2

  Description: The operands can multiplied and stored in the Rd.

- PE Control :

  PECONF, PESEL, PEMODE (Horizontal/Vertical/Circular modes), PEEXE

- DMA Control :

  LDPEPRG: Load maximum16 program data from Program memory to Instruction Decoder in PEs

  LDDFB: Load large amount of processing data for PEs from Data Memory to Data Frame Buffer

  LDPEDATA: Load large amount of processing data for PEs from DFB to Embedded SRAM in PE

  WBREG: Write back processed data in Embedded SRAM in PE to the registers in the ICS_RISC

  WBDFB: Write back processed data in Embedded SRAM in PE to DFB

  WBMEM: Write back processed data in DFB to Data Memory

- Dedicated Instructions

  Not yet decided

1.2 Addressing modes.

- Immediate
- Register
- Loop Buffer(LB) Addressing

High-level Modeling of 3D-SoftChip using SystemC

1 Configurable Array Processor (CAP) Chip

1.1 Processing Element : Standard-PE

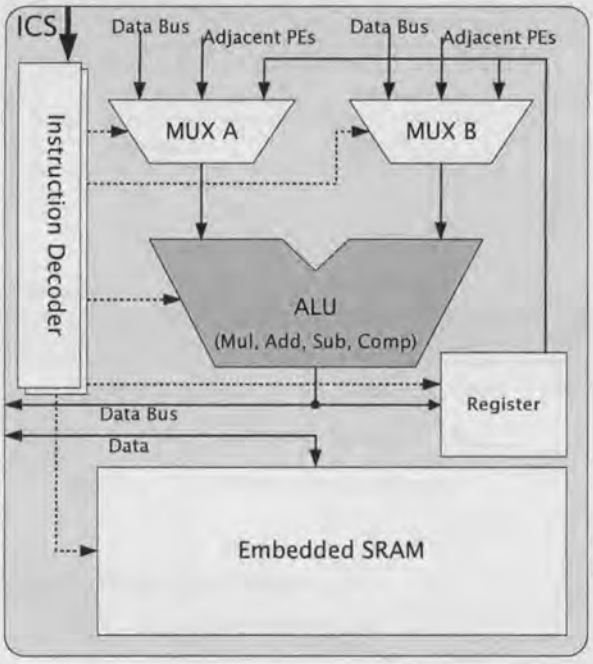


Figure 1.1: Standard-PE architecture

1.2 System Components

- MUX A, MUX B: input operand selection
- Instruction Decoder: 4 sets of ID, each ID have 4 sets of 19-bit registers for S-PE instruction decoding
- ALU : 4-bit ALU with bit-serial multiplier, adder, subtractor, comparator
- Registers : 4 sets of registers
- DourReg : data out register to send data for adjacent PEs(Up/ Down/Left/ Right)
- Embedded SRAM : embedded SRAM (word-length: 4-bit, address : 0~15)

1.3 S-PE functions

Table 1.1: S-PE functions

Function	Mnemonics
A and B	AND
A or B	OR
not A	NOT
A xor B	XOR
A + B	ADD
A – B	SUB
A × B	SPMUL
A comp B	COMP

1.4 S-PE Instruction Format

18	17	16	15 12	11 10	9	8 6	5 3	2 0
WS_en/ RS_en	WR_en/ RR_en	SRAM en	SRAM Selection	Register Selection	DoutR Ctl	SPE_OP	MUX_B	MUX_A

Figure 1.2: S-PE instruction format

1.5 S-PE Block Diagram (In/Output Pin Description)

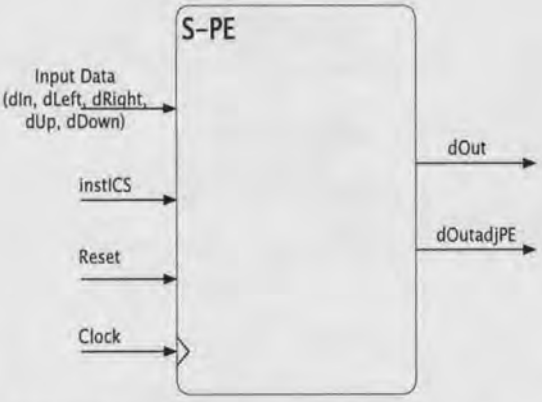


Figure 1.3: S-PE block diagram (Input/Output Pin Description)

1.6 Data-path Architecture of S-PE

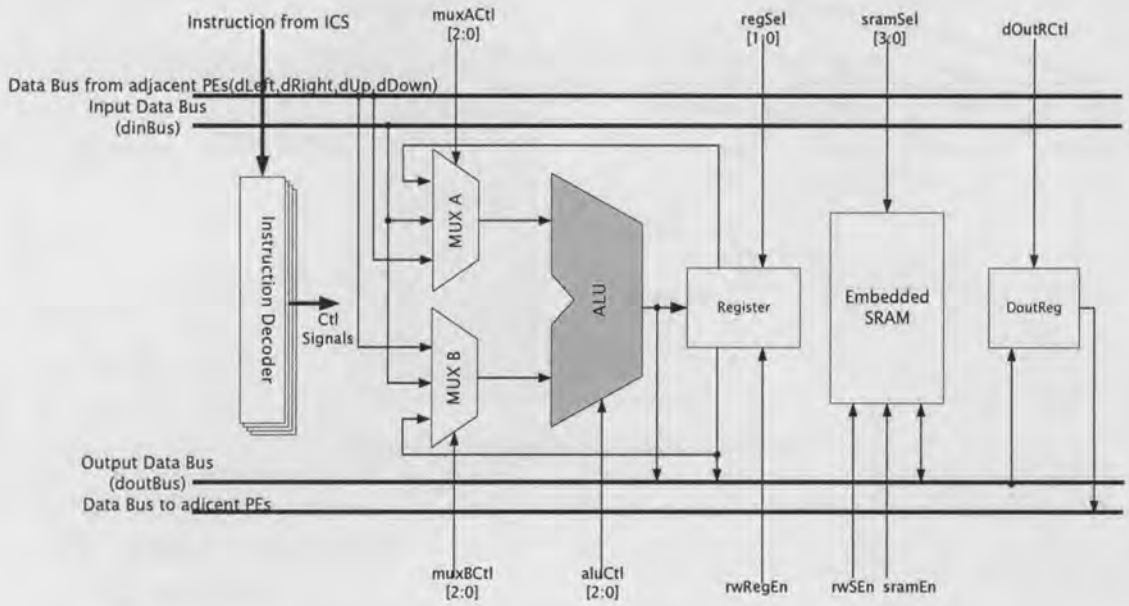


Figure 1.4: Data-path architecture of S-PE

1.7 S-PE Operation Flow

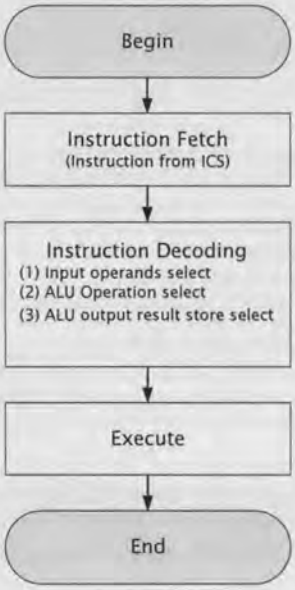


Figure 1.5: S-PE operation flow

1.8 SystemC File Structure

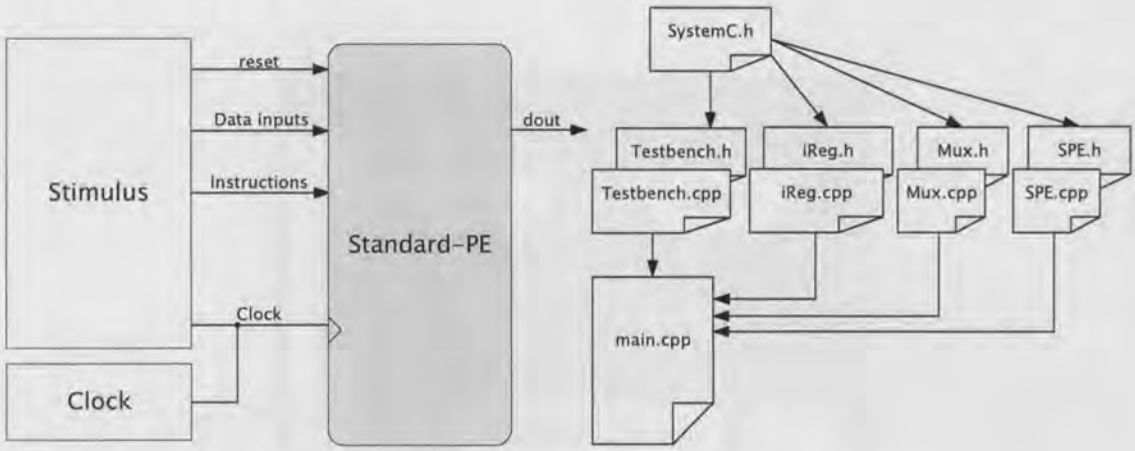


Figure 1.6: SystemC file structure of S-PE

1.9 SystemC Codes for S-PE

See Appendix C

1.10 Output Waveform

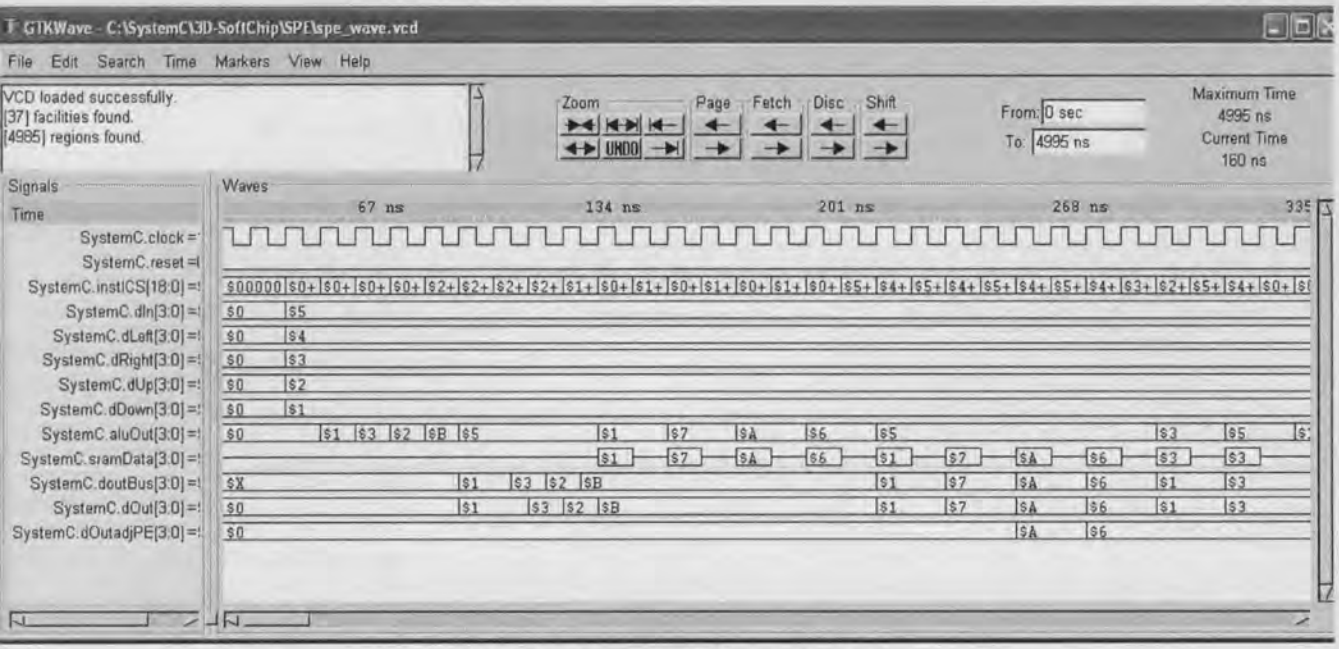


Figure 1.7: Top level simulation result of S-PE

1.11 Processing Element : Processing Accelerator-PE

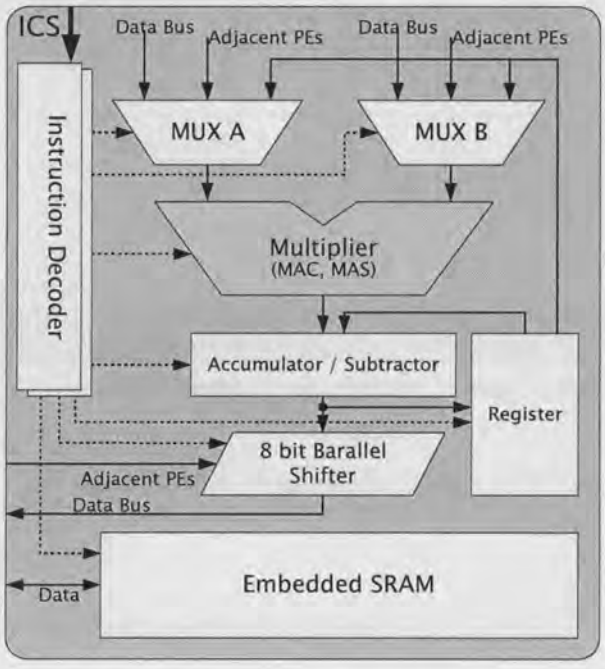


Figure 1.8: Processing Accelerator-PE architecture

1.12 System Components

- MUX A, MUX B : input operand selection
- Instruction Decoder : 4 sets of ID, each ID has 4 sets of 19-bit registers for PA-PE instruction decoding
- Multiplier : a signed 4-bit scalable parallel/parallel multiplier
- Accumulator/Subtractor : to enable MAC, MAS operations within one clock cycle.
- 8-bit Barrel shifter
- Registers : 4 sets of registers.
- Embedded SRAM : embedded SRAM (word-length : 4-bit, address :0~15)

1.13 PA-PE Functions

Table 1.2.PA-PE functions

Function	Mnemonics
A × B	PAMUL
A × B + out(t)	MAC

A × B – out(t)	MAS
Logical Shift Left	LSL
Logical Shift Right	LSR
Arithmetic Shift Right	ASR
Rotate	ROR
|A|(Absolute value)	ABS

1.14 PA-PE Instruction Format

18	17	16	15 12	11 10	9	8 6	5 3	2 0
WS_en/ RS_en	WR_en/ RR_en	SRAM en	SRAM Selection	Register Selection	DoutR Ctl	PA-PE_OP	MUX_B	MUX_A

Figure 1.9: PA-PE instruction format

1.15 PA-PE Block Diagram (In/Output Pin Description)

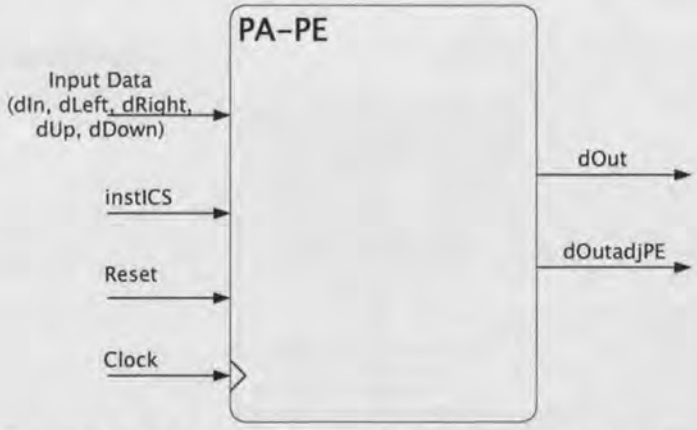


Figure 1.10: PA-PE block diagram (Input/Output Pin Description)

1.16 Data-path Architecture of PA-PE

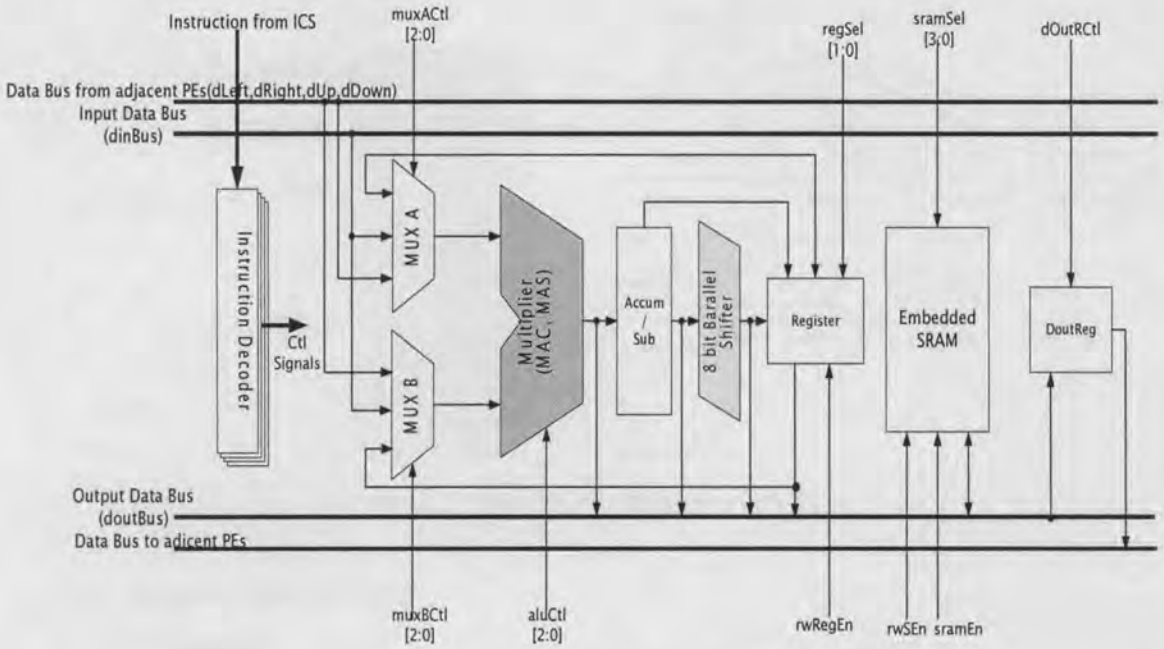


Figure 1.11: Data-path architecture of PA-PE

1.17 PA-PE Operation Flow

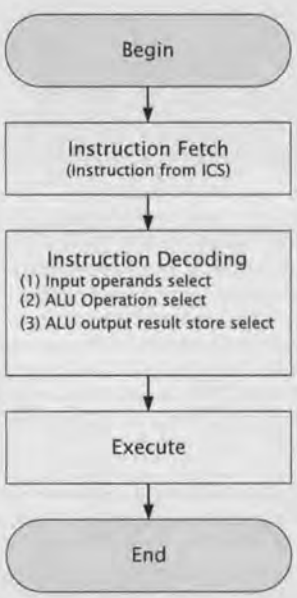


Figure 1.12: PA-PE operation flow

1.18 SystemC File Structure

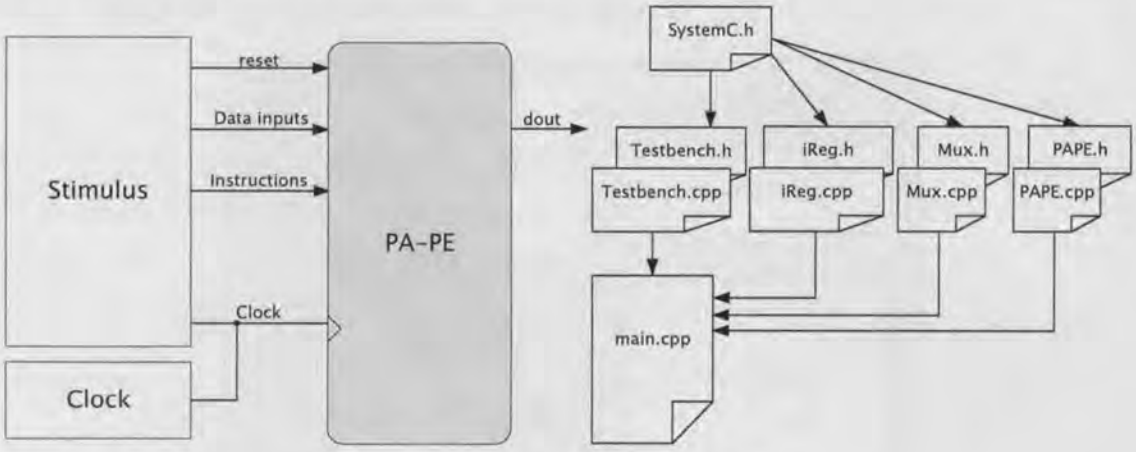


Figure 1.13: SystemC file structure of PA-PE

1.19 SystemC Codes for PA-PE

See Appendix C.

1.20 Output Waveform

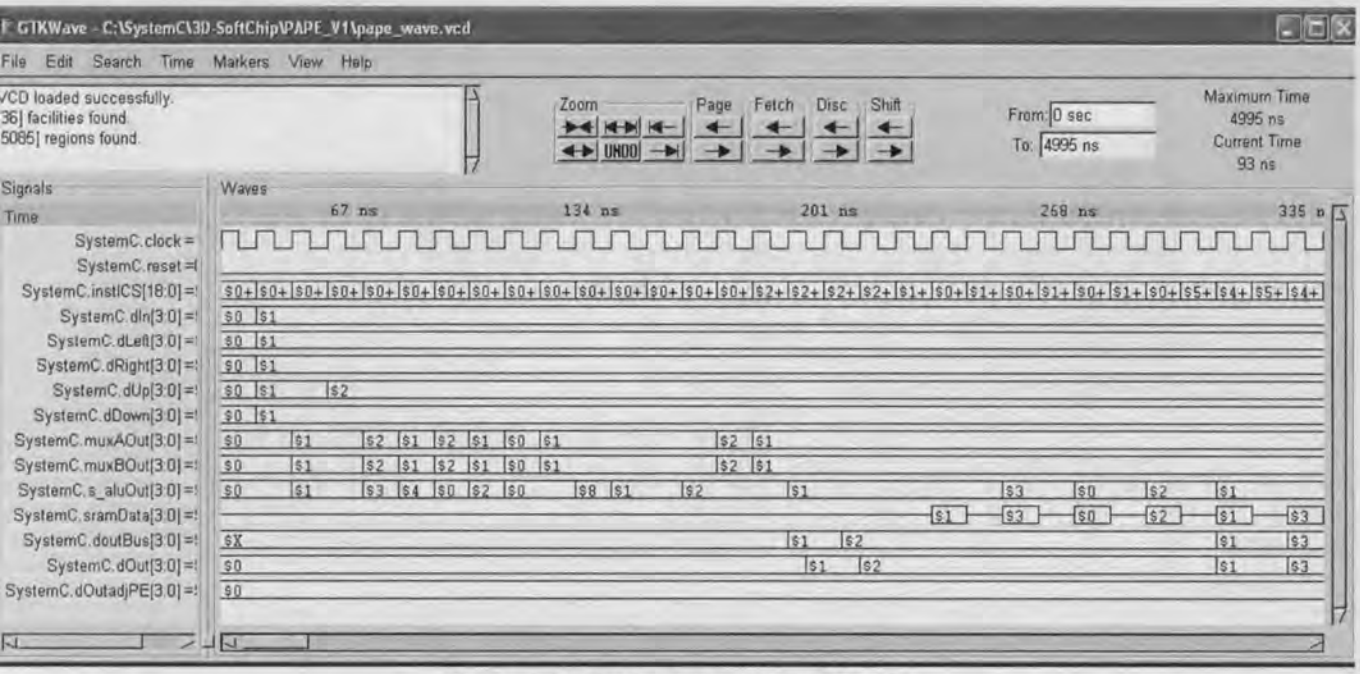


Figure 1.14: Top level simulation of PA-PE

2 ICS(Intelligent Configurable Switch) Chip

2.1 ICS_RISC (32-bit Dedicated RISC Control Processor)

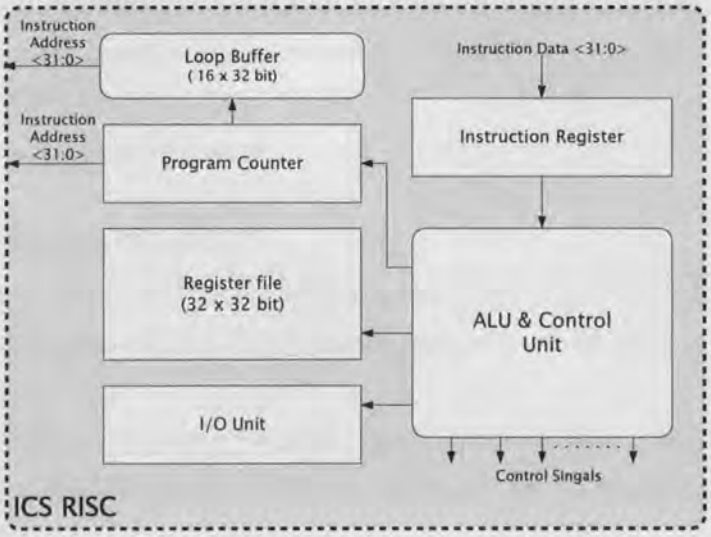


Figure 2.1: Overall architecture of ICS_RISC

2.2 ICS_RISC-Detailed Architecture

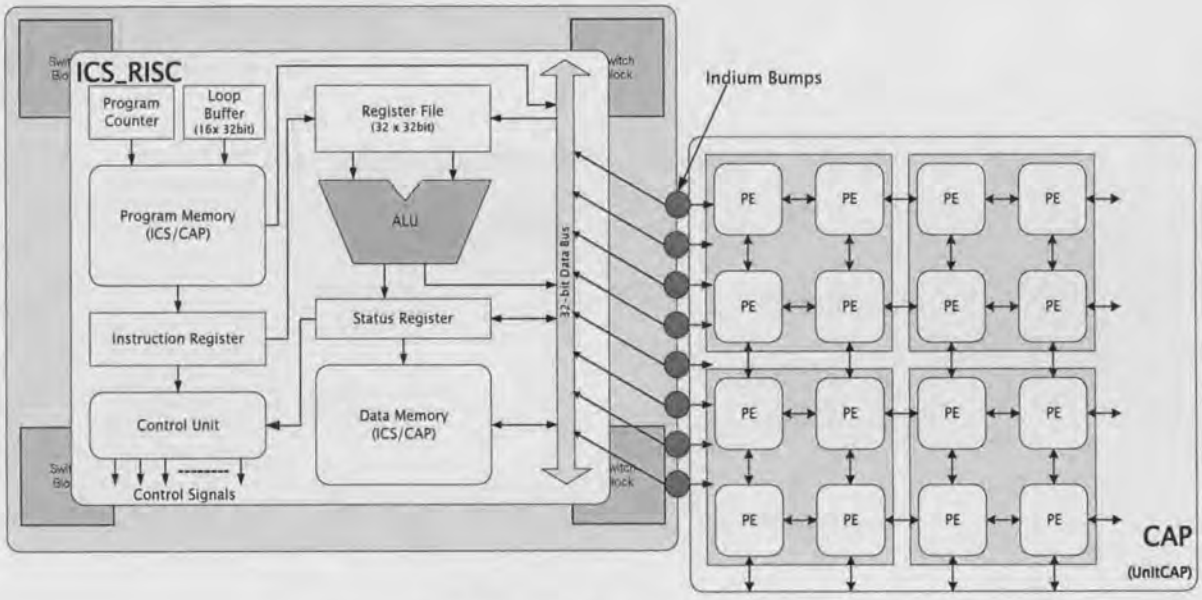


Figure 2.2: Detailed ICS_RISC Architecture

2.3 Special Features (ICS_RISC)

- Harvard architecture, 3 Stage Pipelined architecture(Fetch, Decode, Execute)
- Memory access, during the execution stage, is done by load/store instructions only
- All operations except load/store, PE and DMA operations, are register-to-register within the ICS RISC
- Single-cycle instruction execution

2.4 System Components (ICS_RISC)

- Program Counter : 32th GPR is a program counter
- Loop Buffer : 16 × 32-bit buffer to generate instruction address for iterative characteristic instructions
- Register file(General Purpose Register) : 32 × 32-bit general purpose register
- Status Register : 4 kinds of flags (N: Negative / Less Than, Z: Zero, C: Carry / Borrow, V: Overflow)
- Instruction Register: Instruction decoder for ALU and Control Unit
- ALU & Control Unit: It is consist of ALU, Shifter, Multiplier
- I/O Unit: 32-bit Data input/output register (dInReg, dOutReg)

2.5 ICS RISC Functions

- See Appendix A

2.6 ICS _RISC Block Diagram (Input/Output Pin Description)

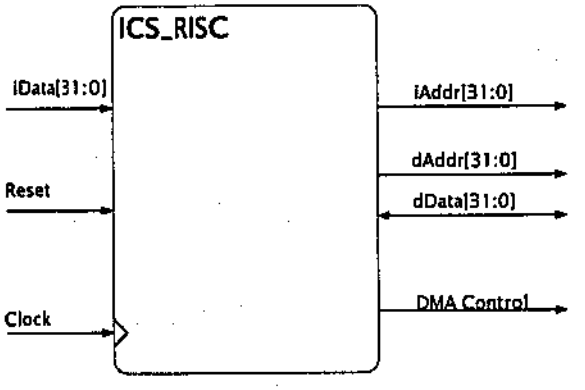


Figure 2.3: ICS_RISC Block Diagra.n (Input/Output Pin Description)

2.7 UnitICS Block Diagram (Input/Output Pin Description)

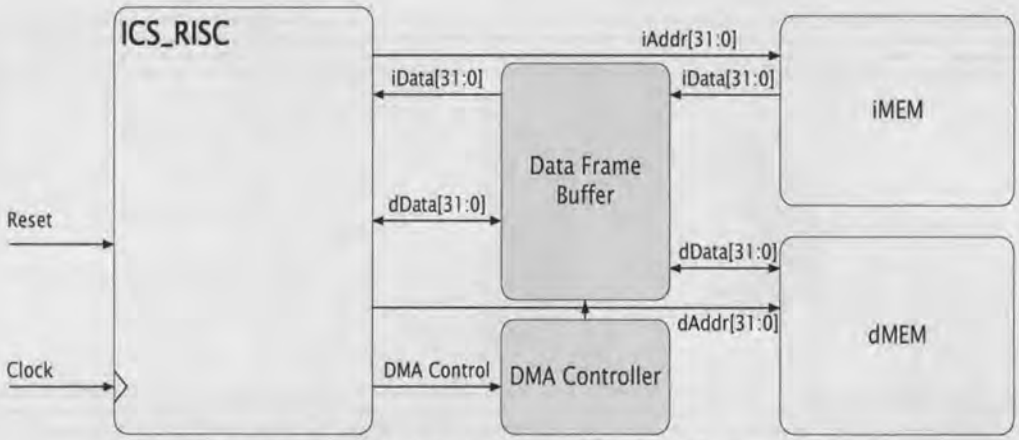


Figure 2.4: UnitICS Block Diagram (Input/Output Pin Description)

2.8 Three-stage Pipeline Architecture (ICS_RISC)

FETCH	DECODE	**EXECUTE**		
	FETCH	**DECODE**	EXECUTE	
		FETCH	DECODE	EXECUTE

Figure 2.5: Three-stage Pipeline Architecture (ICS_RISC)

2.9 Register Architecture (ICS_RISC)

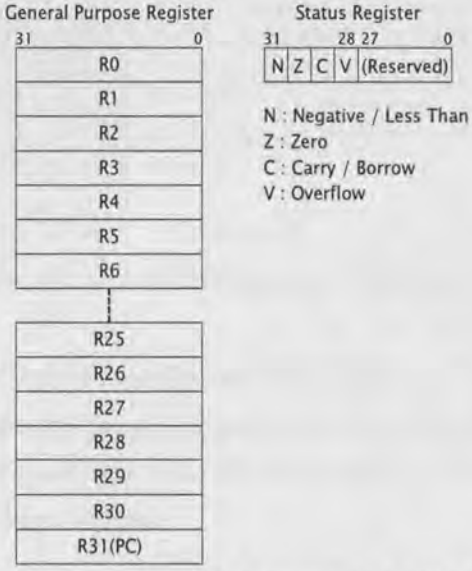


Figure 2.6: Register Architecture (ICS_RISC)

2.10 Data-path Architecture of ICS_RISC

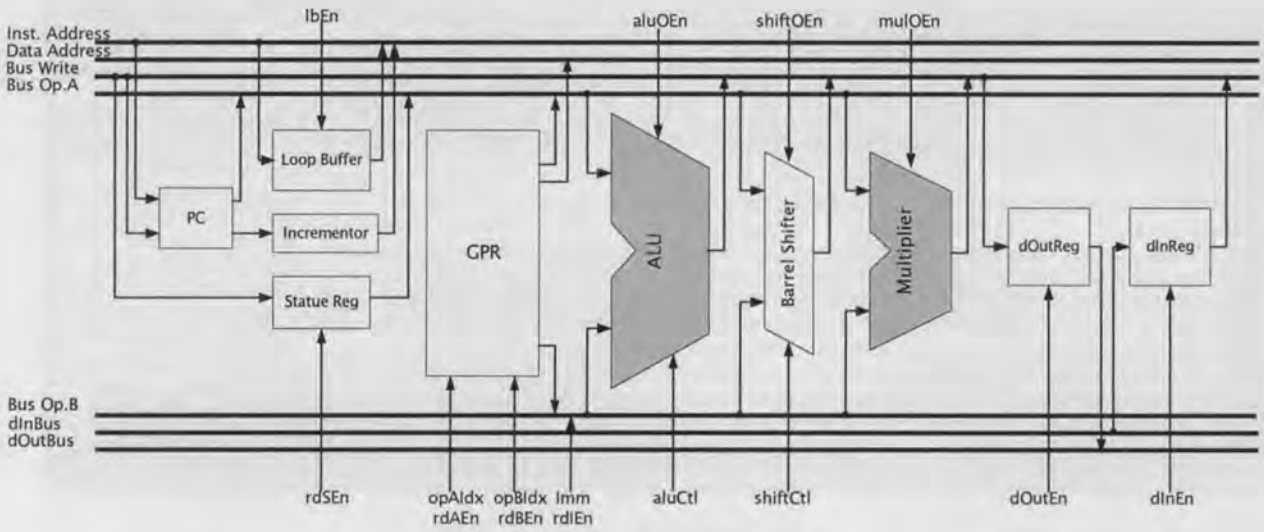


Figure 2.7: Data-path architecture of ICS_RISC

2.11 SystemC File Structure

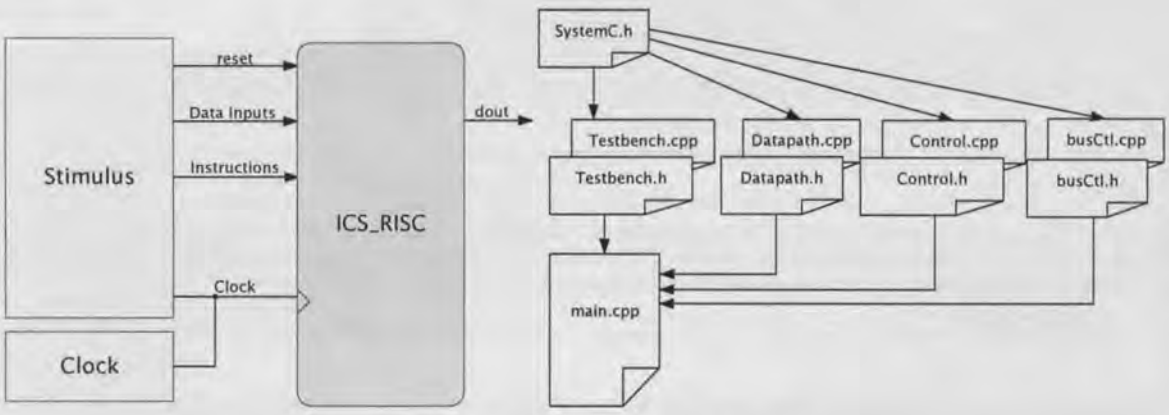


Figure 2.8: SystemC file structure of ICS_RISC

2.12 SystemC Modeling of Data-path Architecture of ICS_RISC

- System Components: (1) Program Counter, (2) Status Register, (3) Loop Buffer, (4) General Purpose Register, (5) ALU, (6) Barrel Shifter, (7) Multiplier, (8) Data Input Register, (9) Data Output Register

- Program Counter (PC)

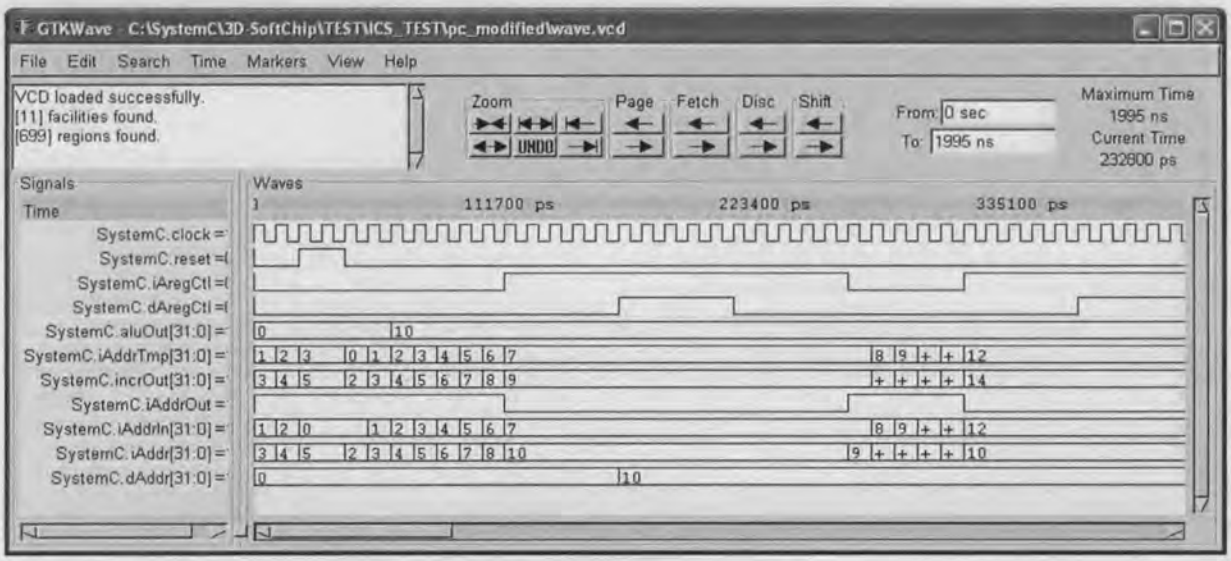


Figure 2.9: Output waveform of PC

- Status Register

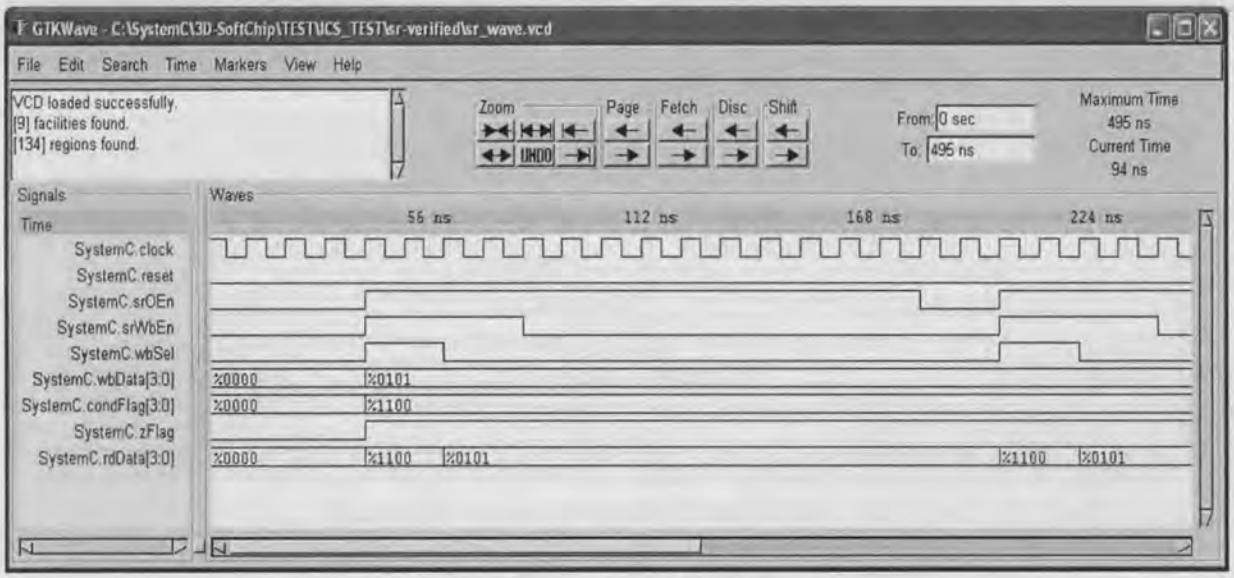


Figure 2.10: Output waveform of Status Register

- Loop Buffer (LB)

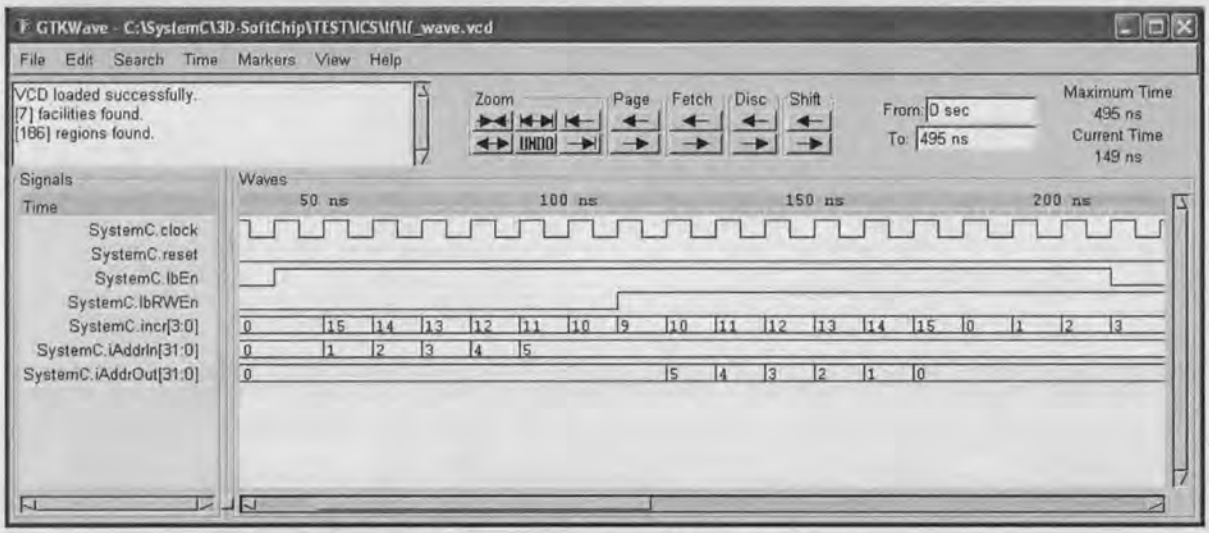


Figure 2.11: Output waveform of Loop Buffer

- General Purpose Register (GPR)

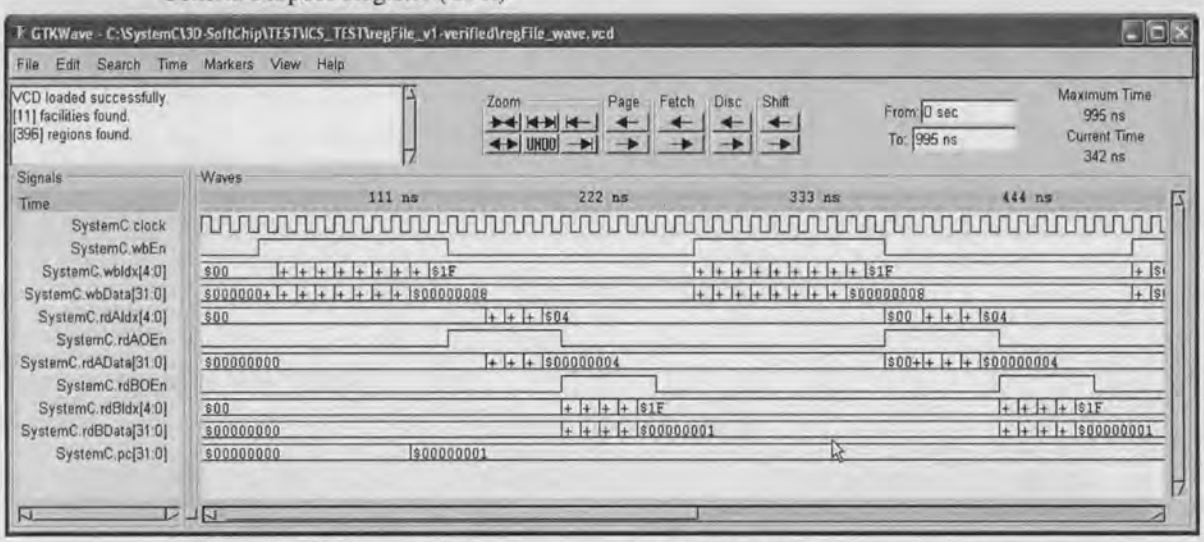


Figure 2.12: Output waveform of Register File

- ALU

Table2.1: ALU Functions

Opcodes				Mnemonics	Description (Immediate)	Description (Register)
0	0	0	0	**MOVA**	Rd = Immediate	Rd = Rs1
0	0	0	1	**MOVB**	Rd = Immediate	Rd = Rs2
0	0	1	0	**AND**	Rd = Rd & Immediate	Rd = Rs1 & Rs2
0	0	1	1	**OR**	Rd = Rd | Immediate	Rd = Rs1 | Rs2
0	1	0	0	**XOR**	Rd = Rd ^ Immediate	Rd = Rs1 ^ Rs2

0	1	0	1	**NOT**	Rd = ~ Immediate	Rd = ~Rs1
0	1	1	0	**ADD**	Rd = Rs1 + Immediate	Rd = Rs1 + Rs2
0	1	1	1	**SUB**	Rd = Rs1 – Immediate	Rd = Rs1 – Rs2
1	0	0	0	**CMP**	Compare Rs1 and Immediate	Compare Rs1 and Rs2
1	0	0	1	**MSR**	Status Register = Immediate	Status Register = Rs1
1	0	1	0	**MRS**	N/A	Rs1 = Status Register

Figure 2.13: Output waveform of ALU

- 32bit Barrel Shifter

Table2.2: Shifter Functions

Shift			Mnemonics	Description
0	0	0	**LSL**	Shift Left
0	0	1	**LSR**	Shift Right
0	1	0	**ASR**	Arithmetic Shift Right
0	1	1	**ROT**	Rotate

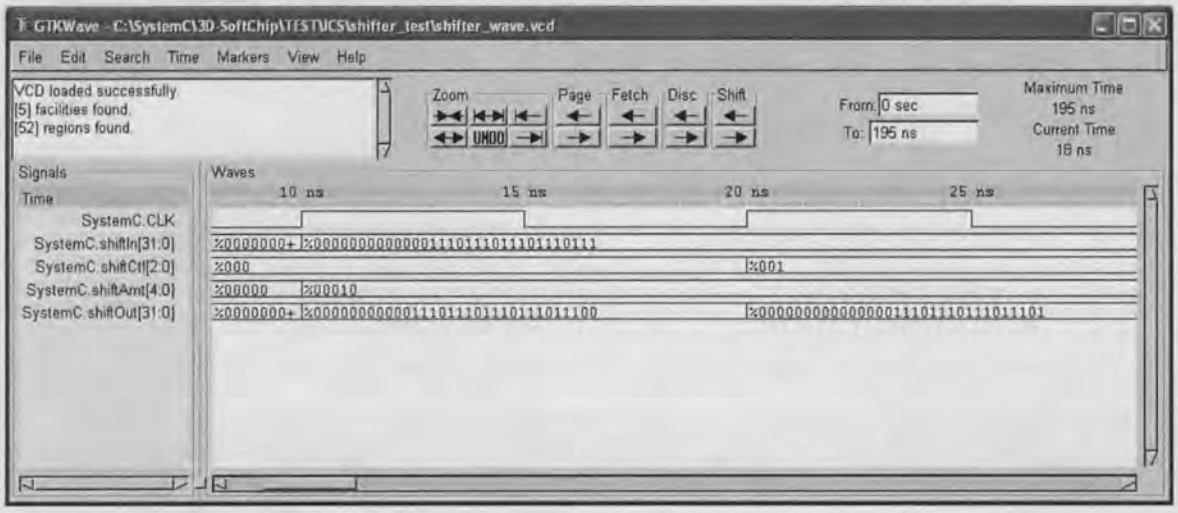


Figure 2.14: Output waveform of 32-bit Barrel Shifter

- 32 × 32 Signed Multiplier

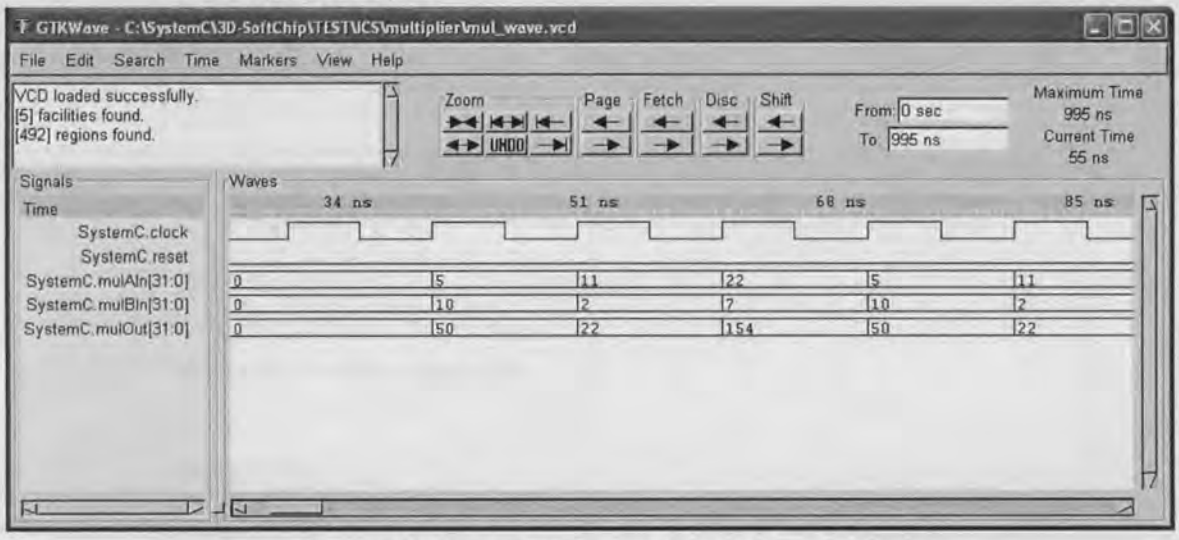


Figure 2.15: Output waveform of Signed 32 × 32 Multiplier

- Data Input Register

- Data Output Register

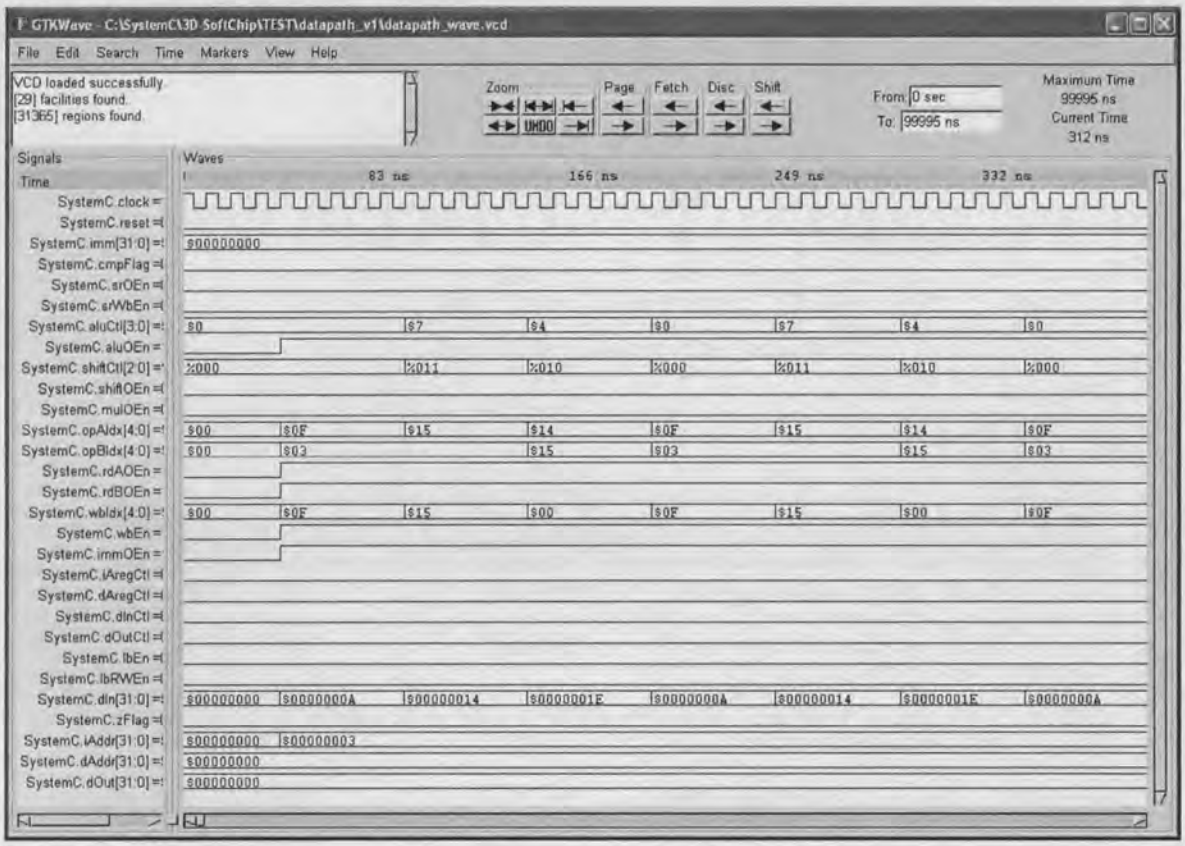


Figure 2.16: Output waveform of top module in data-path architecture

2.13 Control Architecture of ICS_RISC

- Fetch Unit : Fetch the instructions

- Decoder Unit

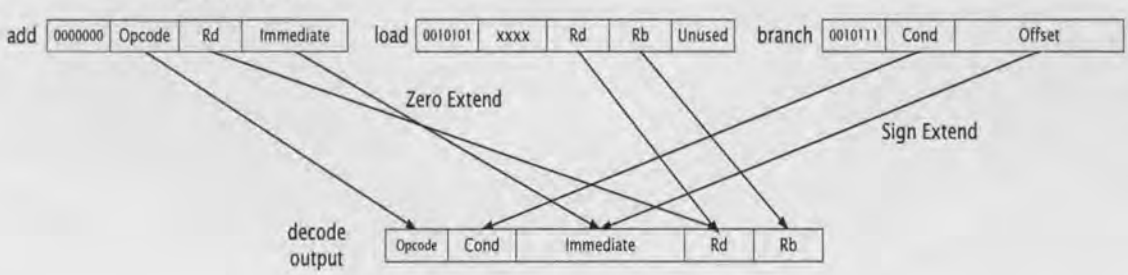


Figure 2.17: Instruction Decoding (1)

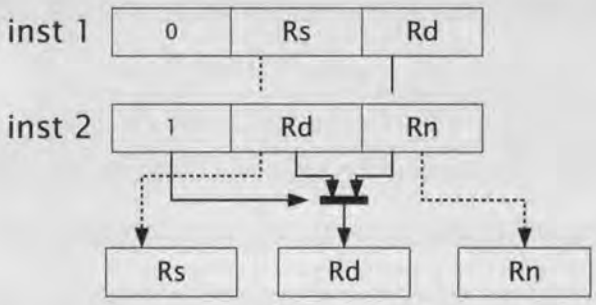


Figure 2.18: Instruction Decoding (2)

Table 2.3: Instruction ID for Instruction Decoding

Instruction ID	Instruction[31:25]	Description
INST_ALUIS	000/0000	ALU Immediate (1 Inst. Word)
INST_ALUIL	000/0001	ALU Immediate (2 Inst. Word)
INST_ALUR	000/0010	ALU Register
INST_ALULB	000/0011	ALU Loop Buffer Addressing
INST_SHRO	001/0100	Shift / Rotate
INST_LOAD	001/0101	Load
INST_STORE	001/0110	Store
INST_BRANCH	001/0111	Branch
INST_PECON	010/1000	PE Control
INST_DMA	1xx/xxxx	DMA Control
INST_MUL	011/1111	Multiply

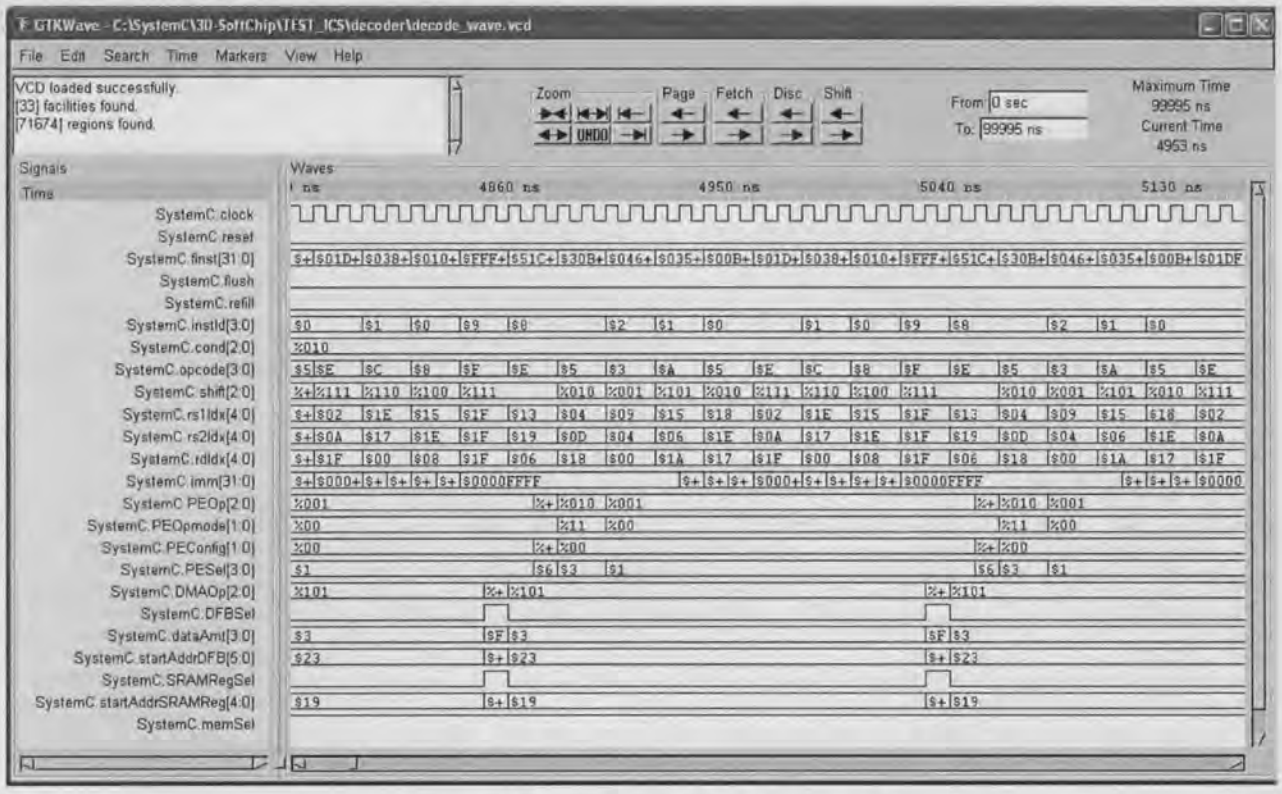


Figure 2.19: Output waveform of Instruction Decoding

- Execute Unit

Table 2.4: Control Signal according to the Instruction

Instruction	ALU Op.	ALU Out	Shifter Out	Mul. Out	Operand A	Operand B
ALU Immediate (1 Inst. Word)	Op Code	Enable	Disable	Disable	Rd	Immediate (4,8,16bit)
ALU Immediate (2 Inst. Word)	Op Code	Enable	Disable	Disable	Rd	Immediate (32bit)
ALU Register	Op Code	Enable	Disable	Disable	Rs1	Rs2
ALU LB Addr.	Op Code	Enable	Disable	Disable	Rs1	Rs2
Shift / Rotate	Don't Care	Disable	Enable	Disable	Rb (Rs1)	ShiftAmt
Load	MOV	Disable	Disable	Disable	Rb (Rs1)	Don't Care
Store	MOV	Disable	Disable	Disable	Rb (Rs1)	Rd
Branch	ADD	Enable	Disable	Disable	PC	Immediate
PE Control	Don't Care	Disable	Disable	Disable	Don't Care	Don't Care
DMA Control	Don't Care	Disable	Disable	Disable	Don't Care	Don't Care
Multiply	Don't Care	Disable	Disable	Enable	Rs1	Rs2

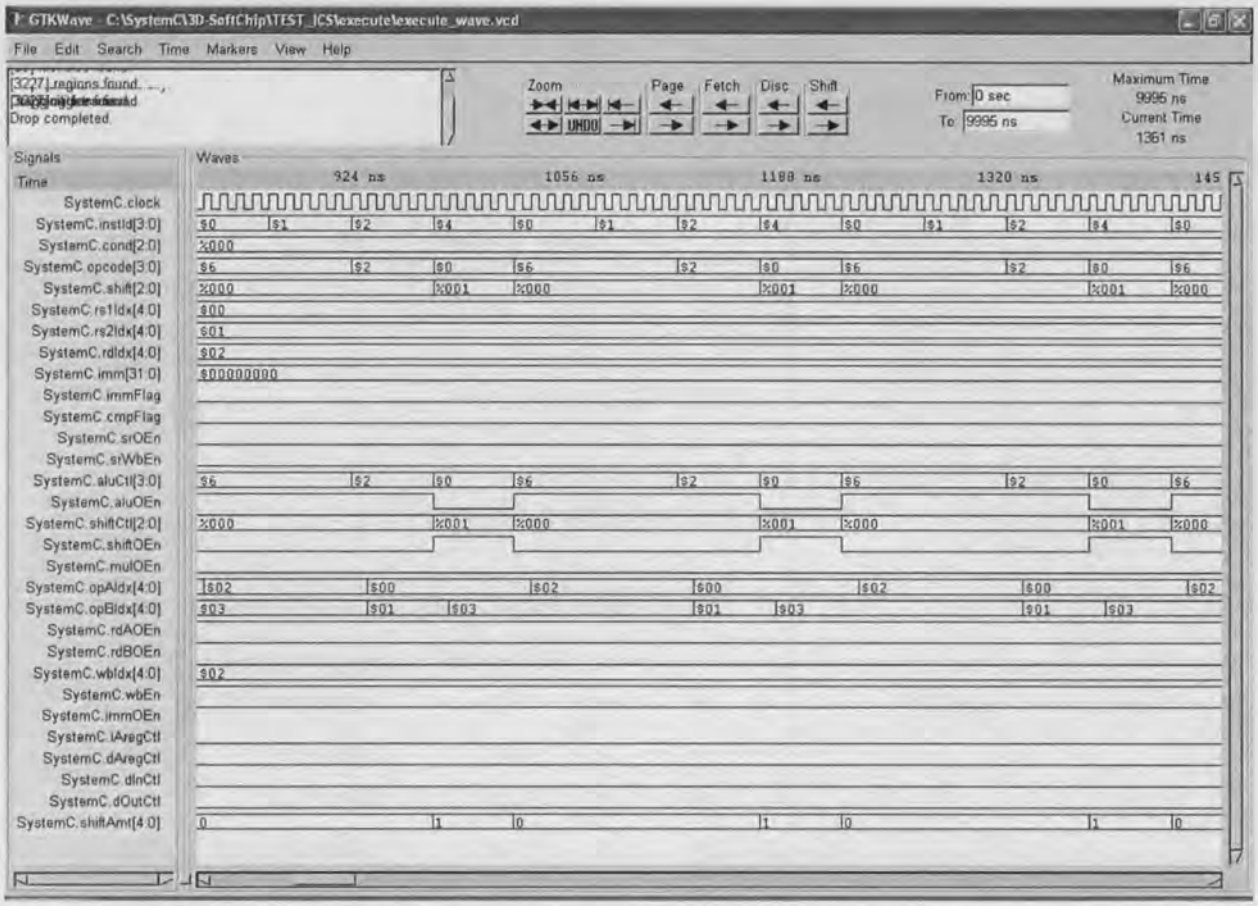


Figure 2.20: Output waveform of Instruction Execution

- Pipeline Register

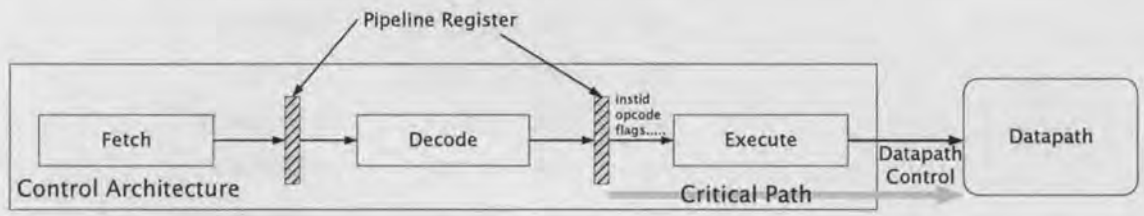


Figure2.21: Conventional Pipeline Register Architecture

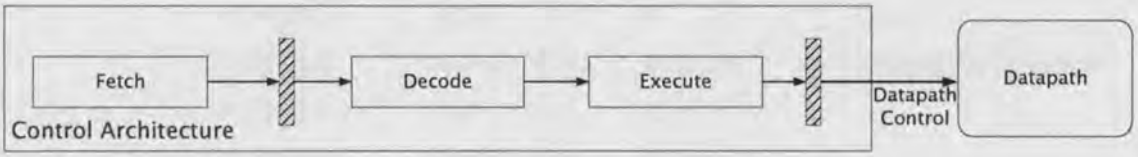


Figure2.22: Modified Pipeline Register Architecture (High-Speed)

- Pipeline Control (reset, flush and refill)

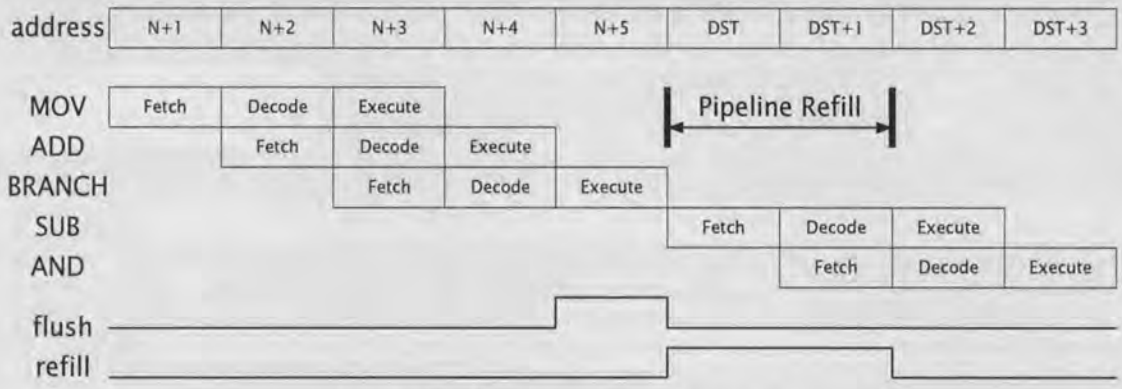


Figure2.23: Branch Instruction Execution

2.14 Top-level Simulation Result of ICS_RISC

- Simple Program for Verification

```
0000/0000   //MOV  R0,   #0          //Simple Loop Program
0001/0001   //MOV  R1,   #1
0002/0002   //MOV  R2,   #2
0003/0003   //MOV  R3,   #3
0004/0004   //MOV  R4,   #4
0005/0005   //MOV  R5,   #5
0006/0006   //MOV  R6,   #6
0007/0007   //MOV  R7,   #7
0408/0000   //MOV  R8,   R0
0409/0020   //MOV  R9,   R1
040A/0040   //MOV  R10,  R2
040B/0060   //MOV  R11,  R3
040C/0080   //MOV  R12,  R4
040D/00A0   //MOV  R13,  R5
040E/00C0   //MOV  R14,  R6
040F/00E0   //MOV  R15,  R7          //End
0450/4280   //AND  R16,  R8 & R9     //Simple ALU Program
0471/52C0   //OR   R17,  R10 | R11
0492/6340   //XOR  R18,  R12 ^ R13
04D3/6B80   //ADD  R19,  R14 + R15
```

04F4/6B80 //SUB R20, R14 – R15 //End

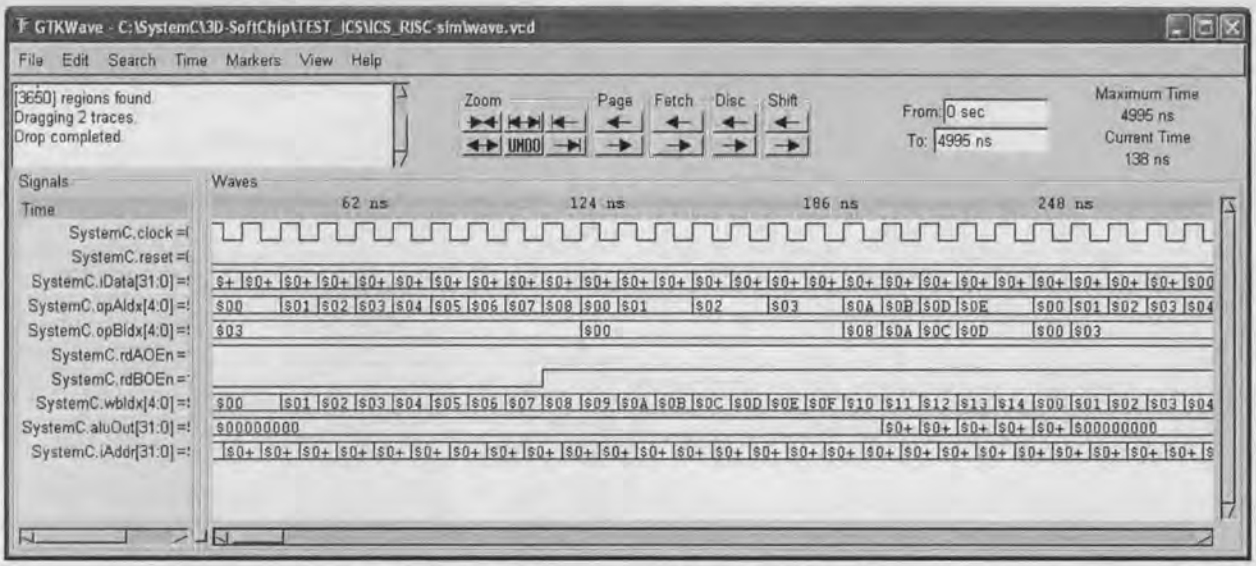


Figure2.24: Top level Simulation Result of ICS_RISC

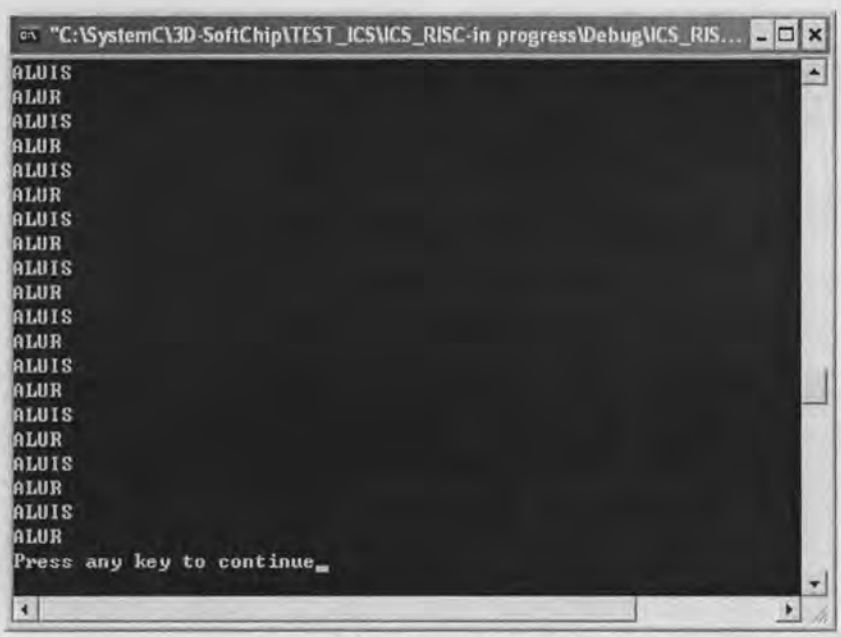


Figure2.25: Instruction Index

3 UnitChip

3.1 SystemC File Structure

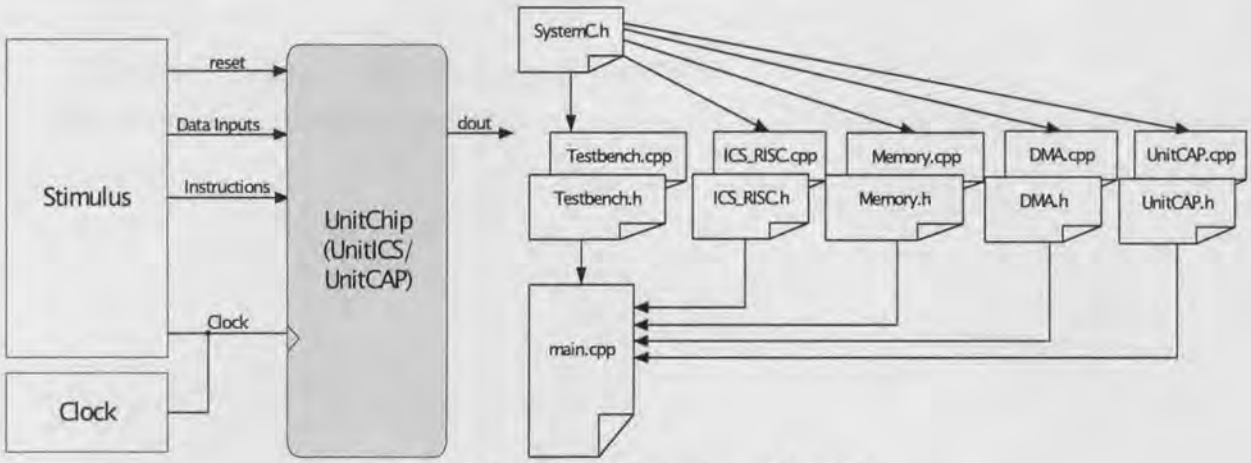


Figure3.1: SystemC file structure of UnitChip

3.2 Top-level Simulation Result of UnitChip

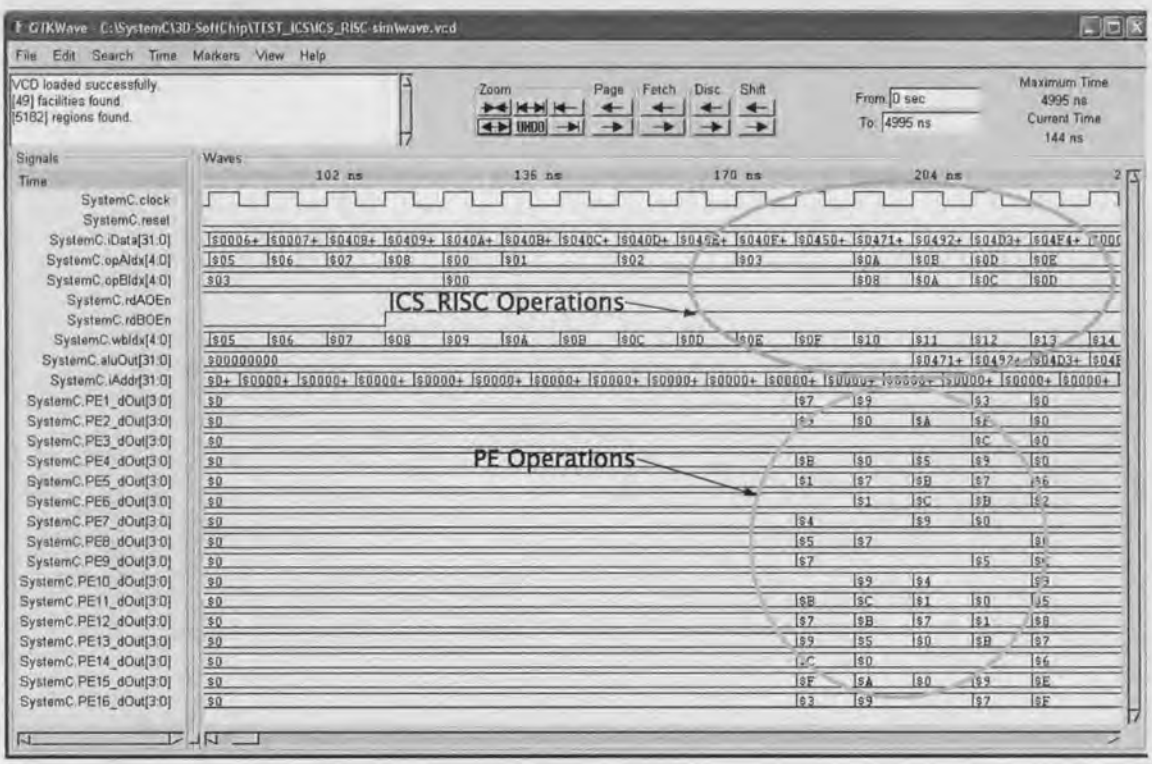


Figure3.2: Top-level Simulation Result of UnitChip

References for the ICS_RISC

[1] Yeong-don Bae, "Basic Microprocessor Design", http://www.donny.co.kr

[2] Yap Zi He, "Building A RISC Microcontroller in an FPGA",
http://www.opencores.org/projects/riscmcu

SystemC Codes

1 Standard-PE.

```
/*
 * iReg: Instruction Reg for Standard-PE(header file for iReg)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: iReg.h
 * Revision history: Version1
 * Date: 17/1/2005
 */

#include "systemc.h"

SC_MODULE(iReg) {
        sc_in<sc_uint<19> >     inst;       //instruction input
        sc_out<sc_uint<3> >     muxACtl;    //muxA Ctl
        sc_out<sc_uint<3> >     muxBCtl;    //muxB Ctl
        sc_out<sc_uint<3> >     sopSel;     //S-PE operation sel
        sc_out<bool>            doutRCtl;   //data-out reg ctl
        sc_out<sc_uint<2> >     regSel;     //internal reg sel
        sc_out<sc_uint<4> >     sramSel;    //SRAM sel
        sc_out<bool>            sramEn;     //SRAM enable signal
        sc_out<bool>            rwRegEn;    //internal reg read/write signal
        sc_out<bool>            rwSEn;      //SRAM read/write enable signal

        void do_iReg();

        SC_CTOR(iReg) {
                SC_METHOD(do_iReg);
                sensitive << inst;
#ifdef SIM
                muxACtl.initialize(0);
                muxBCtl.initialize(0);
                sopSel.initialize(0);
                doutRCtl=0;
                regSel.initialize(0);
                sramSel.initialize(0);
                sramEn=0;
                rwRegEn=0;
                rwSEn=0;
#endif
        }
};

/*
 * iReg: Instruction Reg for Standard-PE(source file for iReg)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: iReg.cpp
 * Revision history : Version1
 * Date: 17/1/2005
 */

#include "iReg.h"

void iReg::do_iReg() {
        sc_uint<19>             tmp_inst;
        tmp_inst = inst.read();

        rwSEn                   = tmp_inst[18];
        rwRegEn                 = tmp_inst[17];
```

```
        sramEn          = tmp_inst[16];
        sramSel         = tmp_inst.range(15,12);
        regSel          = tmp_inst.range(11,10);
        doutRCtl        = tmp_inst[9];
        sopSel          = tmp_inst.range(8,6);
        muxBCtl         = tmp_inst.range(5,3);
        muxACtl         = tmp_inst.range(2,0);
}
```

```
/*
 * Mux: Mux for Standard-PE(header file for Mux)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: mux.h
 * Revision history: Version1
 * Date: 17/1/2005
 */

#include "systemc.h"

SC_MODULE(mux) {
        sc_in<sc_uint<3> >      muxCtl;         //mux ctl input
        sc_in<sc_uint<4> >      dIn;            //input data
        sc_in<sc_uint<4> >      dReg;           //data from internal Reg
        sc_in<sc_uint<4> >      dLeft;          //data from adjacent PE(from left PE)
        sc_in<sc_uint<4> >      dRight;         //data from adjacent PE(from right PE)
        sc_in<sc_uint<4> >      dUp;            //data from adjacent PE(from upside PE);
        sc_in<sc_uint<4> >      dDown;          //data from adjacent PE(from downside PE);
        sc_out<sc_uint<4> >     muxOut;
        sc_out<bool>            dReq;           //data request for internal register

        void do_mux();

        SC_CTOR(mux) {
                SC_METHOD(do_mux);
                sensitive << muxCtl << dIn << dReg << dLeft << dRight << dUp << dDown;
#ifdef SIM
                muxOut.initialize(0);
                dReq=0;
#endif
        }
};

/*
 * Mux: Mux for Standard-PE(source file for Mux)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: mux.cpp
 * Revision history: Version1
 * Date: 17/1/2005
 */

#include "mux.h"

void mux::do_mux() {
        switch (muxCtl.read()) {
                case 0: muxOut = dIn;           dReq=0;   break;
                case 1: muxOut = dReg;          dReq=1;   break;
                case 2: muxOut = dLeft;         dReq=0;   break;
                case 3: muxOut = dRight;        dReq=0;   break;
                case 4: muxOut = dUp;           dReq=0;   break;
                case 5: muxOut = dDown;         dReq=0;   break;
                default:                                  break;
        }
}
```

```
/*
 * SPE: Standard-PE for CAP(Configurable Array Processor)(header file for SPE)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: spe.h
 * Revision history: Version1
 * Date: 17/1/2005
 */

#include "systemc.h"
#include "iReg.h"
#include "mux.h"

SC_MODULE(spe) {
        sc_in<bool>             clock;
        sc_in<bool>             reset;
        sc_in<sc_uint<19> >     instICS;    //instruction input from ICS
        sc_in<sc_uint<4> >      dIn, dLeft, dRight, dUp, dDown;         //data inputs
        sc_out<sc_uint<4> >     dOut;               //data output
        sc_out<sc_uint<4> >     dOutadjPE;          //data output for adjacent PEs

        //temp signal for instruction
        sc_signal<sc_uint<19> > s_inst;

        //temp signals from iReg(Instruction Decoder)
        sc_signal<sc_uint<3> >s_muxACtl;
        sc_signal<sc_uint<3> > s_muxBCtl;
        sc_signal<sc_uint<3> > s_sopSel;
        sc_signal<bool>         s_doutRCtl;
        sc_signal<sc_uint<2> > s_regSel;
        sc_signal<sc_uint<4> > s_sramSel;
        sc_signal<bool>       s_sramEn;
        sc_signal<bool>       s_rwRegEn;
        sc_signal<bool>       s_rwSEn;

        //temp signals for mux in/output and ALU inputs
        sc_signal<sc_uint<4> >s_dIn, s_dLeft, s_dRight, s_dUp, s_dDown;
        sc_signal<sc_uint<4> >dRegOutA;//reg out for muxA input
        sc_signal<sc_uint<4> >dRegOutB;//reg out for muxB input
        sc_signal<sc_uint<4> >muxAOut;
        sc_signal<sc_uint<4> >muxBOut;
        sc_signal<bool>       dReqA, dReqB;         //data request for register

        //temp signal for ALU output
        sc_signal<sc_uint<4> >aluOut;

        //temp signal for internal register
        sc_signal<sc_uint<4> >regIn;
        sc_signal<sc_uint<4> >tmp1, tmp2, tmp3, tmp4;
        sc_signal<sc_uint<4> >regOut;

        //temp signals for SRAM
        sc_signal_rv<4>       sramData;
        sc_lv<4>              ramData[16];

        //temp signal for output data bus
//      sc_signal<sc_uint<4> >doutBus;
        sc_signal<sc_lv<4> >  doutBus;

        void do_latch();
        void do_alu();
        void do_reg();
        void do_sram();
        void do_doutReg();

        iReg*     iReg1;
        iReg*     iReg2;
        iReg*     iReg3;
        iReg*     iReg4;
```

```
        mux*    muxA;
        mux*    muxB;

        SC_CTOR(spe) {
                iReg1=new iReg("iReg");
                iReg1->inst(s_inst);                    iReg1->muxACtl(s_muxACtl);
                iReg1->muxBCtl(s_muxBCtl);              iReg1->sopSel(s_sopSel);
                iReg1->doutRCtl(s_doutRCtl);            iReg1->regSel(s_regSel);
                iReg1->sramSel(s_sramSel);              iReg1->sramEn(s_sramEn);
                iReg1->rwRegEn(s_rwRegEn);              iReg1->rwSEn(s_rwSEn);
                iReg2=new iReg("iReg");
                iReg2->inst(s_inst);                    iReg2->muxACtl(s_muxACtl);
                iReg2->muxBCtl(s_muxBCtl);              iReg2->sopSel(s_sopSel);
                iReg2->doutRCtl(s_doutRCtl);            iReg2->regSel(s_regSel);
                iReg2->sramSel(s_sramSel);              iReg2->sramEn(s_sramEn);
                iReg2->rwRegEn(s_rwRegEn);              iReg2->rwSEn(s_rwSEn);
                iReg3=new iReg("iReg");
                iReg3->inst(s_inst);                    iReg3->muxACtl(s_muxACtl);
                iReg3->muxBCtl(s_muxBCtl);              iReg3->sopSel(s_sopSel);
                iReg3->doutRCtl(s_doutRCtl);            iReg3->regSel(s_regSel);
                iReg3->sramSel(s_sramSel);              iReg3->sramEn(s_sramEn);
                iReg3->rwRegEn(s_rwRegEn);              iReg3->rwSEn(s_rwSEn);
                iReg4=new iReg("iReg");
                iReg4->inst(s_inst);                    iReg4->muxACtl(s_muxACtl);
                iReg4->muxBCtl(s_muxBCtl);              iReg4->sopSel(s_sopSel);
                iReg4->doutRCtl(s_doutRCtl);            iReg4->regSel(s_regSel);
                iReg4->sramSel(s_sramSel);              iReg4->sramEn(s_sramEn);
                iReg4->rwRegEn(s_rwRegEn);              iReg4->rwSEn(s_rwSEn);

                muxA=new mux("mux");
                muxA->muxCtl(s_muxACtl);                muxA->dIn(s_dIn);
                muxA->dReg(dRegOutA);                   muxA->dLeft(s_dLeft);
                muxA->dRight(s_dRight);                 muxA->dUp(s_dUp);
                muxA->dDown(s_dDown);                   muxA->muxOut(muxAOut);
                muxA->dReq(dReqA);
                muxB=new mux("mux");
                muxB->muxCtl(s_muxBCtl);                muxB->dIn(s_dIn);
                muxB->dReg(dRegOutB);                   muxB->dLeft(s_dLeft);
                muxB->dRight(s_dRight);                 muxB->dUp(s_dUp);
                muxB->dDown(s_dDown);                   muxB->muxOut(muxBOut);
                muxB->dReq(dReqB);

                SC_METHOD(do_latch);
                sensitive_pos << clock;
                sensitive_neg << reset;

                SC_METHOD(do_alu);
                sensitive << clock << muxAOut << muxBOut << s_sopSel << s_rwRegEn;

                SC_METHOD(do_reg);
                sensitive << clock << s_regSel << s_rwRegEn << regIn << dReqA << dReqB;

                SC_METHOD(do_sram);
                sensitive << clock << s_rwSEn << s_sramEn << s_sramSel << regIn << sramData;

                SC_METHOD(do_doutReg);
                sensitive << clock << s_doutRCtl << doutBus;

#ifdef SIM
                doutBus.initialize(0);
                dOut.initialize(0);
                dOutadjPE.initialize(0);
                for (int i=0; i<16; i++) ramData[i]="XXXX";
#endif
        }
};
```

```
/*
 * SPE: Standard-PE for CAP(Configurable Array Processor)(source file for SPE)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: spe.cpp
 * Revision history: Version1
 * Date: 17/1/2005
 */

#include "spe.h"

// Latch
void spe::do_latch() {
        if (reset) {
                s_inst.write(0);        s_dIn.write(0);         s_dLeft.write(0);
                s_dRight.write(0);      s_dUp.write(0);         s_dDown.write(0);
        } else {
                s_inst.write(instICS.read());
                s_dIn.write(dIn.read());                //input data bus
                s_dLeft.write(dLeft.read());
                s_dRight.write(dRight.read());
                s_dUp.write(dUp.read());
                s_dDown.write(dDown.read());
        }
}

// ALU
#define comp(a,b) (((a)>(b))?1: (((a)==(b))?0: -1))             //comparator

void spe::do_alu() {
#ifdef SIM
        unsigned short result=0;
#else
        unsigned short result;
#endif
        unsigned short src1=muxAOut.read();
        unsigned short src2=muxBOut.read();

        switch(s_sopSel.read()) {
                case 0: result = src1 & src2;                   break;  //and
                case 1: result = src1 | src2;                   break;  //or
                case 2: result = ~src1;                         break;  //not
                case 3: result = src1 ^ src2;                   break;  //xor
                case 4: result = src1 + src2;                   break;  //add
                case 5: result = src1 - src2;                   break;  //sub
                case 6: result = src1 * src2;                   break;  //spmul
                case 7: result = comp(src1,src2);               break;  //comp
                default:                                        break;
        }
        aluOut.write(result);
        regIn.write(result);
}

// Internal Register
void spe::do_reg() {
        if (s_rwRegEn) {                        //read operation
                switch (s_regSel.read()) {
                        case 0: regOut.write(tmp1);                     break;
                        case 1: regOut.write(tmp2);                     break;
                        case 2:     regOut.write(tmp3);                 break;
                        case 3:     regOut.write(tmp4);                 break;
                        default:                                        break;
                }
                if (dReqA) {                                    //output control
                        dRegOutA = regOut;
                } else {
                        doutBus = sc_lv<4> (regOut);
                        dOut = sc_uint<4> (doutBus);
                }
```

```
                if (dReqB) {
                        dRegOutB = regOut;
                } else {
                        doutBus = sc_lv<4> (regOut);
                        dOut = sc_uint<4> (doutBus);
                }
        } else {                                        //write operation
                switch (s_regSel.read()) {
                        case 0: tmp1 = regIn;                   break;
                        case 1: tmp2 = regIn;                   break;
                        case 2: tmp3 = regIn;                   break;
                        case 3: tmp4 = regIn;                   break;
                        default:                                break;
                }
        }
}

// SRAM
void spe::do_sram() {
        if (s_sramEn) {
                if (s_rwSEn) {                          //read operation
                        sramData.write(ramData[s_sramSel.read()]);
                        doutBus = sramData;
                        dOut = sc_uint<4> (sramData);
//                      dOut = sc_uint<4> (doutBus); //-- dOut has a dummy value(#F)
                } else {                                //write operation
                        sramData= sc_lv<4> (regIn);
                        ramData[s_sramSel.read()] = sramData;
                }
        } else {
                sramData = "ZZZZ";
        }
}

// Data output register
void spe::do_doutReg() {
        if (s_doutRCtl) {
                dOutadjPE = sc_uint<4> (doutBus);
        }
}
```

2 Proeceeing Accelerator-PE.

```
/*
 * iReg: Instruction Reg for Processing Accelerator-PE(header file for iReg)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(cklm@student.ecu.edu.au)
 * File name: iReg.h
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "systemc.h"

SC_MODULE(iReg) {
        sc_in<sc_uint<19> >             inst;           //instruction input
        sc_out<sc_uint<3> >             muxACtl;        //muxA Ctl
        sc_out<sc_uint<3> >             muxBCtl;        //muxB Ctl
        sc_out<sc_uint<3> >             sopSel;         //S-PE operation sel
        sc_out<bool>                    doutRCtl;       //data-out reg ctl
        sc_out<sc_uint<2> >             regSel;         //internal reg sel
        sc_out<sc_uint<4> >             sramSel;        //SRAM sel
        sc_out<bool>                    sramEn;         //SRAM enable signal
        sc_out<bool>                    rwRegEn;        //internal reg read/write signal
        sc_out<bool>                    rwSEn;          //SRAM read/write enable signal

        void do_iReg();

        SC_CTOR(iReg) {
                SC_METHOD(do_iReg);
                sensitive << inst;
#ifdef SIM
                muxACtl.initialize(0);
                muxBCtl.initialize(0);
                sopSel.initialize(0);
                doutRCtl=0;
                regSel.initialize(0);
                sramSel.initialize(0);
                sramEn=0;
                rwRegEn=0;
                rwSEn=0;
#endif
        }
};

/*
 * iReg: Instruction Reg for Processing Accelerator-PE(source file for iReg)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(cklm@student.ecu.edu.au)
 * File name: iReg.cpp
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "iReg.h"

void iReg::do_iReg() {
        sc_uint<19>             tmp_inst;
        tmp_inst = inst.read();

        rwSEn                   = tmp_inst[18];
        rwRegEn                 = tmp_inst[17];
        sramEn                  = tmp_inst[16];
        sramSel                 = tmp_inst.range(15,12);
        regSel                  = tmp_inst.range(11,10);
        doutRCtl                = tmp_inst[9];
        sopSel                  = tmp_inst.range(8,6);
        muxBCtl                 = tmp_inst.range(5,3);
        muxACtl                 = tmp_inst.range(2,0);
}
```

```
/*
 * Mux: Mux for Processing Accelerator-PE(header file for Mux)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: mux.h
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "systemc.h"

SC_MODULE(mux) {
        sc_in<sc_uint<3> >      muxCtl;         //mux ctl input
        sc_in<sc_uint<4> >      dIn;            //input data
        sc_in<sc_uint<4> >      dReg;           //data from internal Reg
        sc_in<sc_uint<4> >      dLeft;          //data from adjacent PE(from left PE)
        sc_in<sc_uint<4> >      dRight;         //data from adjacent PE(from right PE)
        sc_in<sc_uint<4> >      dUp;            //data from adjacent PE(from upside PE);
        sc_in<sc_uint<4> >      dDown;          //data from adjacent PE(from downside PE);
        sc_out<sc_uint<4> >     muxOut;
        sc_out<bool>                    dReq;           //data request for internal register

        void do_mux();

        SC_CTOR(mux) {
                SC_METHOD(do_mux);
                sensitive << muxCtl << dIn << dReg << dLeft << dRight << dUp << dDown;
#ifdef SIM
                muxOut.initialize(0);
                dReq=0;
#endif
        }
};

/*
 * Mux: Mux for Processing Accelerator-PE(source file for Mux)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: mux.cpp
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "mux.h"

void mux::do_mux() {
        switch (muxCtl.read()) {
                case 0: muxOut = dIn;           dReq=0; break;
                case 1: muxOut = dReg;          dReq=1; break;
                case 2: muxOut = dLeft;         dReq=0; break;
                case 3: muxOut = dRight;        dReq=0; break;
                case 4: muxOut = dUp;           dReq=0; break;
                case 5: muxOut = dDown;         dReq=0; break;
                default:                                break;
        }
}
```

```
/*
 * ALU: ALU for Processing Accelerator-PE(header file for ALU)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: alu.h
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "systemc.h"
```

```
SC_MODULE(alu) {
        sc_in<sc_uint<4> >    aluAIn;
        sc_in<sc_uint<4> >    aluBIn;
        sc_in<sc_uint<3> >    aluCtl;
        sc_out<sc_uint<4> >   aluOut;
        sc_out<sc_uint<4> >   regIn;
        sc_out<sc_uint<4> >   regTmp;          //temp reg for MAC,MAS & output for test

        void do_alu();

        SC_CTOR(alu) {
                SC_METHOD(do_alu);
                sensitive << aluAIn << aluBIn << aluCtl;
#ifdef SIM
                aluOut.initialize(0);
                regIn.initialize(0);
                regTmp.initialize(0);
#endif
        }
};

/*
 * ALU: ALU for Processing Accelerator-PE(source file for ALU)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: alu.cpp
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "alu.h"

//#define MAC(A,B,P)  (((A)*(B))+(P))                 //mac
//#define MAS(A,B,P)  (((A)*(B))-(P))                 //mas
//#define    //asr-when the data-type is signed, it should be modified
#define    ROR(A)    ((((A&0x0f)&0x1)?(((A&0x0f)>>0x1)|0x8):(A&0x0f)>>0x1)&0x0f) //rotate
#define    ABS(A)    (((A&0x0f)<0x0?(-1*(A&0x0f)):(A&0x0f))&0x0f) //abs

void alu::do_alu() {
        sc_uint<4>   result,src1,src2,tmp,mulTmp;          //temp signals
        src1 = aluAIn.read();
        src2 = aluBIn.read();

        switch (aluCtl.read()) {
                case 0: result = src1*src2;                                  break;    //pamul
                case 1: mulTmp = src1*src2;
                        result = mulTmp + sc_uint<4> (regTmp);               break;  //mac
                case 2: mulTmp = src1*src2;
                        result = mulTmp - sc_uint<4> (regTmp);               break;    //mas
                case 3: result = src1<<1;                                    break;  //lsl
                case 4: result = src1>>1;                                    break;  //lsr
                //when the data-type is signed, it should be modified(asr)
                case 5: result = src1>>1;                                    break;  //asr(divider/2)
                case 6: result = ROR(src1);                                  break;  //ror
                //when the data-type is signed, it can be applied(abs)
                case 7: result = ABS(src1);                                  break;  //abs
                default:                                                     break;
        }
        aluOut.write(result);
        regIn.write(result);
        tmp = aluOut;
        regTmp.write(tmp);    //defined in the header file, signal for Test
        //sc_out<sc_uint4> > regTmp;
}
```

```
/*
 * PAPE: Processing Accelerator-PE for CAP(header file for PAPE)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: pape.h
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "systemc.h"
#include "iReg.h"
#include "mux.h"
#include "alu.h"

SC_MODULE(pape) {
        sc_in<bool>                     clock;
        sc_in<bool>                     reset;
        sc_in<sc_uint<19> >             instICS;    //instruction input from ICS
        sc_in<sc_uint<4> >              dIn, dLeft, dRight, dUp, dDown;          //data inputs
        sc_out<sc_uint<4> >             dOut;                   //data output
        sc_out<sc_uint<4> >             dOutadjPE;              //data output for adjacent PEs

        //temp signal for instruction
        sc_signal<sc_uint<19> >         s_inst;

        //temp signals from iReg(Instruction Decoder)
        sc_signal<sc_uint<3> >s_muxACtl;
        sc_signal<sc_uint<3> >  s_muxBCtl;
        sc_signal<sc_uint<3> >  s_sopSel;
        sc_signal<bool>         s_doutRCtl;
        sc_signal<sc_uint<2> >  s_regSel;
        sc_signal<sc_uint<4> >  s_sramSel;
        sc_signal<bool>         s_sramEn;
        sc_signal<bool>         s_rwRegEn;
        sc_signal<bool>         s_rwSEn;

        //temp signals for mux in/output and ALU inputs
        sc_signal<sc_uint<4> >s_dIn, s_dLeft, s_dRight, s_dUp, s_dDown;
        sc_signal<sc_uint<4> >dRegOutA;//reg out for muxA input
        sc_signal<sc_uint<4> >dRegOutB;//reg out for muxB input
        sc_signal<sc_uint<4> >muxAOut;
        sc_signal<sc_uint<4> >muxBOut;
        sc_signal<bool>         dReqA, dReqB;       //data request for register

        //temp signal for ALU output
        sc_signal<sc_uint<4> >s_aluOut;
        sc_signal<sc_uint<4> >s_regTmp;

        //temp signal for internal register
        sc_signal<sc_uint<4> >s_regIn;
        sc_signal<sc_uint<4> >tmp1, tmp2, tmp3, tmp4;
        sc_signal<sc_uint<4> >regOut;

        //temp signals for SRAM
        sc_signal_rv<4>         sramData;
        sc_lv<4>                ramData[16];

        //temp signal for output data bus
//      sc_signal<sc_uint<4> >doutBus;
        sc_signal<sc_lv<4> >    doutBus;

        void do_latch();
        void do_reg();
        void do_sram();
        void do_doutReg();

        iReg*       iReg1;
        iReg*       iReg2;
        iReg*       iReg3;
```

```
        iReg*   iReg4;
        mux*    muxA;
        mux*    muxB;
        alu*    aluPAPE;

        SC_CTOR(pape) {
                iReg1=new iReg("iReg");
                iReg1->inst(s_inst);            iReg1->muxACtl(s_muxACtl);
                iReg1->muxBCtl(s_muxBCtl);      iReg1->sopSel(s_sopSel);
                iReg1->doutRCtl(s_doutRCtl);    iReg1->regSel(s_regSel);
                iReg1->sramSel(s_sramSel);      iReg1->sramEn(s_sramEn);
                iReg1->rwRegEn(s_rwRegEn);      iReg1->rwSEn(s_rwSEn);
                iReg2=new iReg("iReg");
                iReg2->inst(s_inst);            iReg2->muxACtl(s_muxACtl);
                iReg2->muxBCtl(s_muxBCtl);      iReg2->sopSel(s_sopSel);
                iReg2->doutRCtl(s_doutRCtl);    iReg2->regSel(s_regSel);
                iReg2->sramSel(s_sramSel);      iReg2->sramEn(s_sramEn);
                iReg2->rwRegEn(s_rwRegEn);      iReg2->rwSEn(s_rwSEn);
                iReg3=new iReg("iReg");
                iReg3->inst(s_inst);            iReg3->muxACtl(s_muxACtl);
                iReg3->muxBCtl(s_muxBCtl);      iReg3->sopSel(s_sopSel);
                iReg3->doutRCtl(s_doutRCtl);    iReg3->regSel(s_regSel);
                iReg3->sramSel(s_sramSel);      iReg3->sramEn(s_sramEn);
                iReg3->rwRegEn(s_rwRegEn);      iReg3->rwSEn(s_rwSEn);
                iReg4=new iReg("iReg");
                iReg4->inst(s_inst);            iReg4->muxACtl(s_muxACtl);
                iReg4->muxBCtl(s_muxBCtl);      iReg4->sopSel(s_sopSel);
                iReg4->doutRCtl(s_doutRCtl);    iReg4->regSel(s_regSel);
                iReg4->sramSel(s_sramSel);      iReg4->sramEn(s_sramEn);
                iReg4->rwRegEn(s_rwRegEn);      iReg4->rwSEn(s_rwSEn);

                muxA=new mux("mux");
                muxA->muxCtl(s_muxACtl);        muxA->dIn(s_dIn);
                muxA->dReg(dRegOutA);           muxA->dLeft(s_dLeft);
                muxA->dRight(s_dRight);         muxA->dUp(s_dUp);
                muxA->dDown(s_dDown);           muxA->muxOut(muxAOut);
                muxA->dReq(dReqA);
                muxB=new mux("mux");
                muxB->muxCtl(s_muxBCtl);        muxB->dIn(s_dIn);
                muxB->dReg(dRegOutB);           muxB->dLeft(s_dLeft);
                muxB->dRight(s_dRight);         muxB->dUp(s_dUp);
                muxB->dDown(s_dDown);           muxB->muxOut(muxBOut);
                muxB->dReq(dReqB);

                aluPAPE=new alu("alu");
                aluPAPE->aluAIn(muxAOut);       aluPAPE->aluBIn(muxBOut);
                aluPAPE->aluCtl(s_sopSel);      aluPAPE->aluOut(s_aluOut);
                aluPAPE->regIn(s_regIn);        aluPAPE->regTmp(s_regTmp);

                SC_METHOD(do_latch);
                sensitive_pos << clock;
                sensitive_neg << reset;

                SC_METHOD(do_reg);
                sensitive << clock << s_regSel << s_rwRegEn << s_regIn << dReqA << dReqB;

                SC_METHOD(do_sram);
                sensitive << clock << s_rwSEn << s_sramEn << s_sramSel << s_regIn << sramData;

                SC_METHOD(do_doutReg);
                sensitive << clock << s_doutRCtl << doutBus;

#ifdef SIM
                doutBus.initialize(0);
                dOut.initialize(0);
                dOutadjPE.initialize(0);
                for (int i=0; i<16; i++) ramData[i]="XXXX";
#endif
        }
};
```

```
/*
 * PAPE: Processing Accelerator-PE for CAP(source file for PAPE)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: pape.cpp
 * Revision history: Version1
 * Date: 29/1/2005
 */

#include "pape.h"

// Latch
void pape::do_latch() {
        if (reset) {
                s_inst.write(0);        s_dIn.write(0);         s_dLeft.write(0);
                s_dRight.write(0);      s_dUp.write(0);         s_dDown.write(0);
        } else {
                s_inst.write(instICS.read());
                s_dIn.write(dIn.read());                //input data bus
                s_dLeft.write(dLeft.read());
                s_dRight.write(dRight.read());
                s_dUp.write(dUp.read());
                s_dDown.write(dDown.read());
        }
}

// Internal Register
void pape::do_reg() {
        if (s_rwRegEn) {                //read operation
                switch (s_regSel.read()) {
                        case 0: regOut.write(tmp1);                     break;
                        case 1: regOut.write(tmp2);                     break;
                        case 2:  regOut.write(tmp3);                    break;
                        case 3:  regOut.write(tmp4);                    break;
                        default:                                        break;
                }
                if (dReqA) {                                    //output control
                        dRegOutA = regOut;
                } else {
                        doutBus = sc_lv<4> (regOut);
                        dOut = sc_uint<4> (doutBus);
                }

                if (dReqB) {
                        dRegOutB = regOut;
                } else {
                        doutBus = sc_lv<4> (regOut);
                        dOut = sc_uint<4> (doutBus);
                }
        } else if (s_rwRegEn==0) {              //write operation
                        switch (s_regSel.read()) {
                                case 0: tmp1 = s_regIn;         break;
                                case 1: tmp2 = s_regIn;         break;
                                case 2: tmp3 = s_regIn;         break;
                                case 3: tmp4 = s_regIn;         break;
                                default:                        break;
                        }
        }
}

// SRAM
void pape::do_sram() {
        if (s_sramEn) {
                if (s_rwSEn) {                  //read operation
                        sramData.write(ramData[s_sramSel.read()]);
                        doutBus = sramData;
                        dOut = sc_uint<4> (sramData);
//                      dOut = sc_uint<4> (doutBus); //-- dOut has a dummy value(#F)
                } else {                        //write operation
                        sramData= sc_lv<4> (s_regIn);
```

```
                        ramData[s_sramSel.read()] = sramData;
                }
        } else {
                sramData = "ZZZZ";
        }
}

// Data output register
void pape::do_doutReg() {
        if (s_doutRCtl) {
                dOutadjPE = sc_uint<4> (doutBus);
        }
}
```

3 ICS_RISC

3.1 Datapath Architecture

```
/*
 * PC: Program Counter
 * for ICS(Intelligent Configurable Switch)RISC Core(header file for pc)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: pc.h
 * Revision history: Version1
 * Date: 23/3/2005
 */

#include "systemc.h"

SC_MODULE(pc) {
        sc_in<bool>             clock;
        sc_in<bool>             reset;
        sc_in<bool>             iAregCtl;   //Select Signal between aluOut/incrOut
        sc_in<bool>             dAregCtl;
        sc_in<sc_uint<32> >     aluOut;
        sc_out<sc_uint<32> >    iAddr;      //Instruction Address
        sc_out<sc_uint<32> >    dAddr;      //Data Address

        void do_pc();
        void do_autoIncr();

        sc_uint<32>             iAddrTmp;
        sc_uint<32>             incrOut;
        bool                    iAddrOut;
        sc_uint<32>             iAddrIn;

        SC_CTOR(pc) {
                SC_METHOD(do_pc);
                sensitive << clock.pos() << reset << iAregCtl << dAregCtl << aluOut;
                SC_METHOD(do_autoIncr);
                sensitive << clock.pos() << reset;
#ifdef SIM
                iAddr.initialize(0);
                dAddr.initialize(0);
#endif
        }
};


/*
 * PC: Program Counter
 * for ICS(Intelligent Configurable Switch)RISC Core(source file for pc)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
```

```
 * File name: pc.cpp
 * Revision history: Version1
 * Date: 23/3/2005
 */

#include "pc.h"

void pc::do_pc() {
        bool    iAddrOutTmp;
        if (reset) {
                iAddr = 0;
                dAddr = 0;
        } else {
                iAddrTmp = iAddrIn;
                incrOut = iAddrTmp + 2;
        }
        if (iAregCtl) {
                iAddr = aluOut;
                iAddrOutTmp = 0;
        } else {
                iAddr = incrOut;
                iAddrOutTmp = 1;
        }
        iAddrOut = iAddrOutTmp;

        if (dAregCtl) {
                dAddr = aluOut;
        }
}

void pc::do_autoIncr() {
        if (reset) {
                iAddrIn = 0;
        } else if (iAddrOut) {
                if (clock.posedge()) {
                        iAddrIn++;
                }
        }
}
```

```
/*
 * SR: Status Register
 * for ICS(Intelligent Configurable Switch)RISC Core(header file for sr)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: sr.h
 * Revision history: Version1
 * Date: 3/2/2005
 */

#include "systemc.h"

SC_MODULE(sr) {
        sc_in<bool>                     clock;
        sc_in<bool>                     reset;
        sc_in<sc_uint<4> >              condFlag;
        sc_in<sc_uint<4> >              wbData;
        sc_in<bool>                     wbSel;
        sc_in<bool>                     srOEn;
        sc_in<bool>                     srWbEn;
        sc_out<bool>                    zFlag;
        sc_out<sc_uint<4> >             rdData;

        void do_sr();

        sc_uint<4>                      srData;
```

```
        SC_CTOR(sr) {
                SC_METHOD(do_sr);
                sensitive << clock << reset << condFlag << wbData
                  << wbSel << srOEn << srWbEn;
#ifdef SIM
                zFlag.initialize(0);
                rdData.initialize(0);
#endif
        }
};


/*
 * SR: Status Register
 * for ICS(Intelligent Configurable Switch)RISC Core(source file for sr)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: sr.cpp
 * Revision history: Version1
 * Date: 3/2/2005
 */

#include "sr.h"

void sr::do_sr() {
        if (reset) {
                srData = 0;
        } else if (srWbEn) {
                if (wbSel) {
                        srData = condFlag;
                } else {
                        srData = wbData;
                }
        }

        zFlag = srData[2];

        if (srOEn) {
                rdData = srData;
        }

//      rdData = srOEn ? sc_lv<4> (srData) : "ZZZZ";
}
```

```
/*
 * LF: Loop Buffer
 * for ICS(Intelligent Configurable Switch)RISC Core(header file for lf)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: lf.h
 * Revision history: Version1
 * Date: 14/3/2005
 */

#include "systemc.h"

SC_MODULE(lf) {
        sc_in<bool>             clock;
        sc_in<bool>             reset;
        sc_in<bool>             lbEn;       //Loop Buffer Enable
        sc_in<bool>             lbRWEn;     //Loop Buffer Read/Write Enable
        sc_in<sc_uint<32> >     iAddrIn;    //iAddr Input for LF
        sc_out<sc_uint<32> >    iAddrOut;   //iAddr      Output for LF
//      sc_out<sc_uint<4> >     incr;

        void do_lf();
```

```
        sc_uint<32> buff[16];
        sc_uint<4>          incrTmp;

        SC_CTOR(lf) {
                SC_METHOD(do_lf);
                sensitive << clock.pos() << reset;
#ifdef SIM
                clock=0;
                reset=1;
                lbEn=0;
                lbRWEn=0;
                lAddrIn.initialize(0);
                incrTmp.initialize(15);
#endif
        }
};


/*
 * LF: Loop Buffer
 * for ICS(Intelligent Configurable Switch)RISC Core(source file for lf)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: lf.cpp
 * Revision history: Version1
 * Date: 17/3/2005
 */

#include "lf.h"

void lf::do_lf() {
        if (clock.posedge()) {
                if (lbEn) {
                        if (lbRWEn) {                   //read operation
                                incrTmp++;
                                lAddrOut.write(buff[incrTmp]);
                        } else {                        //write operation
                                incrTmp--;
                                buff[incrTmp] = lAddrIn;
                        }
                }
        }
//      incr = incrTmp;
}
```

```
/*
 * RegFile: 32 x 32 Register file
 * for ICS(Intelligent Configurable Switch)RISC Core(header file for regFile)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: regFile.h
 * Revision history: Version1
 * Date: 3/2/2005
 */

#include "systemc.h"

SC_MODULE(regFile) {
        sc_in<bool>             clock;
        sc_in<sc_uint<5> >      rdAIdx;         //read index A
        sc_in<sc_uint<5> >      rdBIdx;         //read index B
        sc_in<bool>             rdAOEn;         //read A output enable
        sc_in<bool>             rdBOEn;         //read B output enable
        sc_in<sc_uint<5> >      wbIdx;          //writeback index
        sc_in<sc_uint<32> >     wbData;         //writeback data
        sc_in<bool>             wbEn;           //writeback enable
        sc_in<sc_uint<32> >     pc;
```

```
        sc_out<sc_uint<32> > rdAData;   //read data A
        sc_out<sc_uint<32> > rdBData;   //read data B

        sc_signal<sc_uint<32> >gpr0,gpr1,gpr2,gpr3,gpr4,gpr5,gpr6,gpr7,gpr8,gpr9,
                gpr10,gpr11,gpr12,gpr13,gpr14,gpr15,gpr16,gpr17,gpr18,gpr19,gpr20,
                gpr21,gpr22,gpr23,gpr24,gpr25,gpr26,gpr27,gpr28,gpr29,gpr30,gpr31;

        sc_signal<sc_uint<32> >         rdADataTmp, rdBDataTmp;

        void do_regFile();

        SC_CTOR(regFile) {
                SC_METHOD(do_regFile);
                sensitive << clock.pos() << rdAIdx << rdBIdx << rdAOEn << rdBOEn << wbIdx
                                << wbData << wbEn << pc;
#ifdef SIM
                rdAData.initialize(0);
                rdBData.initialize(0);
#endif
                }
};


/*
 * RegFile: 32 x 32 Register file
 * for ICS(Intelligent Configurable Switch)RISC Core(source file for regFile)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: regFile.cpp
 * Revision history: Version1
 * Date: 3/2/2005
 */

#include "regFile.h"

void regFile::do_regFile() {
        if (wbEn) {
                switch (wbIdx.read()) {
                        case 0: gpr0.write(wbData);             break;
                        case 1: gpr1.write(wbData);             break;
                        case 2: gpr2.write(wbData);             break;
                        case 3: gpr3.write(wbData);             break;
                        case 4: gpr4.write(wbData);             break;
                        case 5: gpr5.write(wbData);             break;
                        case 6: gpr6.write(wbData);             break;
                        case 7: gpr7.write(wbData);             break;
                        case 8: gpr8.write(wbData);             break;
                        case 9: gpr9.write(wbData);             break;
                        case 10: gpr10.write(wbData);           break;
                        case 11: gpr11.write(wbData);           break;
                        case 12: gpr12.write(wbData);           break;
                        case 13: gpr13.write(wbData);           break;
                        case 14: gpr14.write(wbData);           break;
                        case 15: gpr15.write(wbData);           break;
                        case 16: gpr16.write(wbData);           break;
                        case 17: gpr17.write(wbData);           break;
                        case 18: gpr18.write(wbData);           break;
                        case 19: gpr19.write(wbData);           break;
                        case 20: gpr20.write(wbData);           break;
                        case 21: gpr21.write(wbData);           break;
                        case 22: gpr22.write(wbData);           break;
                        case 23: gpr23.write(wbData);           break;
                        case 24: gpr24.write(wbData);           break;
                        case 25: gpr25.write(wbData);           break;
                        case 26: gpr26.write(wbData);           break;
                        case 27: gpr27.write(wbData);           break;
                        case 28: gpr28.write(wbData);           break;
                        case 29: gpr29.write(wbData);           break;
                        case 30: gpr30.write(wbData);           break;
                        case 31: gpr31.write(pc);               break;  //for PC
```

```
                default:                        break;
        }
}
if (rdAOEn) {
        switch (rdAIdx.read()) {
                case 0: rdAData = gpr0;         break;
                case 1: rdAData = gpr1;         break;
                case 2: rdAData = gpr2;         break;
                case 3: rdAData = gpr3;         break;
                case 4: rdAData = gpr4;         break;
                case 5: rdAData = gpr5;         break;
                case 6: rdAData = gpr6;         break;
                case 7: rdAData = gpr7;         break;
                case 8: rdAData = gpr8;         break;
                case 9: rdAData = gpr9;         break;
                case 10: rdAData = gpr10;       break;
                case 11: rdAData = gpr11;       break;
                case 12: rdAData = gpr12;       break;
                case 13: rdAData = gpr13;       break;
                case 14: rdAData = gpr14;       break;
                case 15: rdAData = gpr15;       break;
                case 16: rdAData = gpr16;       break;
                case 17: rdAData = gpr17;       break;
                case 18: rdAData = gpr18;       break;
                case 19: rdAData = gpr19;       break;
                case 20: rdAData = gpr20;       break;
                case 21: rdAData = gpr21;       break;
                case 22: rdAData = gpr22;       break;
                case 23: rdAData = gpr23;       break;
                case 24: rdAData = gpr24;       break;
                case 25: rdAData = gpr25;       break;
                case 26: rdAData = gpr26;       break;
                case 27: rdAData = gpr27;       break;
                case 28: rdAData = gpr28;       break;
                case 29: rdAData = gpr29;       break;
                case 30: rdAData = gpr30;       break;
                case 31: rdAData = pc;          break; //for PC
                default:                        break;
        }
}
if (rdBOEn) {
        switch (rdBIdx.read()) {
                case 0: rdBData = gpr0;         break;
                case 1: rdBData = gpr1;         break;
                case 2: rdBData = gpr2;         break;
                case 3: rdBData = gpr3;         break;
                case 4: rdBData = gpr4;         break;
                case 5: rdBData = gpr5;         break;
                case 6: rdBData = gpr6;         break;
                case 7: rdBData = gpr7;         break;
                case 8: rdBData = gpr8;         break;
                case 9: rdBData = gpr9;         break;
                case 10: rdBData = gpr10;       break;
                case 11: rdBData = gpr11;       break;
                case 12: rdBData = gpr12;       break;
                case 13: rdBData = gpr13;       break;
                case 14: rdBData = gpr14;       break;
                case 15: rdBData = gpr15;       break;
                case 16: rdBData = gpr16;       break;
                case 17: rdBData = gpr17;       break;
                case 18: rdBData = gpr18;       break;
                case 19: rdBData = gpr19;       break;
                case 20: rdBData = gpr20;       break;
                case 21: rdBData = gpr21;       break;
                case 22: rdBData = gpr22;       break;
                case 23: rdBData = gpr23;       break;
                case 24: rdBData = gpr24;       break;
                case 25: rdBData = gpr25;       break;
                case 26: rdBData = gpr26;       break;
                case 27: rdBData = gpr27;       break;
```

```
                    case 28: rdBData = gpr28;          break;
                    case 29: rdBData = gpr29;          break;
                    case 30: rdBData = gpr30;          break;
                    case 31: rdBData = pc;             break;    //for PC
                    default:                           break;
                }
        }
}
```

```
/*
 * aluDef: Definition of the ALU functions
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: aluDef.h
 * Revision history: Version1
 * Date: 2/2/2005
 */

#ifndef _ALU_DEFINE_H_
#define _ALU_DEFINE_H_

// ALU Function Definitions
#define CMD_MOVA  0x0
#define CMD_MOVB  0x1
#define CMD_AND         0x2
#define CMD_OR          0x3
#define CMD_XOR         0x4
#define CMD_NOT         0x5
#define CMD_ADD         0x6
#define CMD_SUB         0x7
#define CMD_CMP         0x8

#endif
```

```
/*
 * aluICS: ALU for ICS(Intelligent Configurable Switch)RISC Core(header file for aluICS)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: aluICS.h
 * Revision history: Version1
 * Date: 2/2/2005
 */

#include "systemc.h"
#include "aluDef.h"

SC_MODULE(aluICS) {
        sc_in<sc_uint<32> >     aluAIn;
        sc_in<sc_uint<32> >     aluBIn;
        sc_in<sc_uint<4> >      aluCtl;
//      sc_in<bool>             cIn;            //carry input
        sc_out<sc_uint<32> >    aluOut;
        sc_out<sc_uint<4> >     condFlag;       //conditional flags

        void do_alu();

        sc_signal<bool>         cf, vf, nf, zf;

        SC_CTOR(aluICS) {
                SC_METHOD(do_alu);
                sensitive << aluAIn << aluBIn << aluCtl;
#ifdef SIM
                aluOut.initialize(0);
                condFlag.initialize(0);
#endif
```

```
        }
};

/*
 * aluICS: ALU for ICS(Intelligent Configurable Switch)RISC Core(source file for aluICS)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: aluICS.cpp
 * Revision history: Version1
 * Date: 2/2/2005
 */

#include "aluICS.h"

#define comp(a,b) (((a)>(b))?1: (((a)==(b))?0:-1))

// ALU
void aluICS::do_alu() {
#ifdef SIM
        signed short result = 0;
#else
        signed short result;
#endif
        signed short src1 = aluAIn.read();
        signed short src2 = aluBIn.read();
//      signed short tcin = cIn.read();
        signed short cmd  = aluCtl.read();

        sc_uint<4> tmpCond;

        switch (cmd & 0xF) {
                case 0: result = src1;                  break;  //CMD_MOVA
                case 1: result = src2;                  break;  //CMD_MOVB
                case 2: result = src1 & src2;           break;  //CMD_AND
                case 3: result = src1 | src2;           break;  //CMD_OR
                case 4: result = src1 ^ src2;           break;  //CMD_XOR
                case 5: result = ~src1;                 break;  //CMD_NOT
                case 6: result = src1 + src2;           break;  //CMD_ADD
                case 7: result = src1 - src2;           break;  //CMD_SUB
                case 8: result = comp(src1,src2);       break;  //CMD_CMP
                default:                                break;
        }

// Conditional Flags
        if (result & 0xFFFF0000)        cf.write(1); //carry/Borrow flag
        else        cf.write(0);
        if (result & 0xFFFF0000)        vf.write(1); //overflow flag
        else        vf.write(0);
        result &= 0xFFFF;
        if (result == 0)                zf.write(1); //zero flag
        else        zf.write(0);
        if (result & 0x8000)            nf.write(1);//negative flag
        else        nf.write(0);

        aluOut.write(result);

        tmpCond[3]=nf;
        tmpCond[2]=zf;
        tmpCond[1]=cf;
        tmpCond[0]=vf;
        condFlag=tmpCond;

}
```

```
/*
 * MUL: multiplier
 * for ICS(Intelligent Configurable Switch)RISC Core(header file for multiplier)
```

```
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: mul.h
 * Revision history: Version1
 * Date: 14/3/2005
 */

#include "systemc.h"

SC_MODULE(mul) {
        sc_in<bool>                 clock;
        sc_in<bool>                 reset;
        sc_in<sc_uint<32> >         mulAIn;
        sc_in<sc_uint<32> >         mulBIn;
        sc_out<sc_uint<32> >        mulOut;

        void do_mul();

        SC_CTOR(mul) {
                SC_METHOD(do_mul);
                sensitive << clock << reset << mulAIn << mulBIn;
#ifdef SIM
                mulOut.initialize(0);
#endif
        }
};

/*
 * MUL: multiplier
 * for ICS(Intelligent Configurable Switch)RISC Core(source file for multiplier)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: mul.cpp
 * Revision history: Version1
 * Date: 14/3/2005
 */


#include "mul.h"

void mul::do_mul() {
        sc_uint<32> src1, src2, result;
        src1 = mulAIn;
        src2 = mulBIn;

        result = src1 * src2;

        mulOut.write(result);

}
```

```
/*
 * Shifter: Shifter for ICS(Intelligent Configurable Switch)RISC Core(header file for Shifter)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: shifter.h
 * Revision history: Version1
 * Date: 2/2/2005
 */

#include "systemc.h"

SC_MODULE(shifter) {
        sc_in<sc_uint<32> >         shiftIn;        //shifter input
        sc_in<sc_uint<5> >          shiftAmt;       //shifter amount
        sc_in<sc_uint<3> >          shiftCtl;       //control input
        sc_out<sc_uint<32> >        shiftOut;       //shifter output
```

```
        void do_shift();

        SC_CTOR(shifter) {
                SC_METHOD(do_shift);
                sensitive << shiftIn << shiftAmt << shiftCtl;
#ifdef SIM
                shiftOut.initialize(0);
#endif
        }
};


/*
 * Shifter: Shifter for ICS(Intelligent Configurable Switch)RISC Core(source file for Shifter)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: shifter.cpp
 * Revision history: Version1
 * Date: 2/2/2005
 */

#include "shifter.h"

void shifter::do_shift() {
        sc_uint<32> w_shiftIn,w_shiftOut;
        sc_uint<5> w_shiftAmt;
        sc_uint<3> w_shiftCtl;

        w_shiftIn=shiftIn;
        w_shiftAmt=shiftAmt;
        w_shiftCtl=shiftCtl;

        switch (w_shiftCtl) {
                case 0: w_shiftOut = w_shiftIn << w_shiftAmt; break;     //logical shift left
                case 1: w_shiftOut = w_shiftIn >> w_shiftAmt; break; //logical shift right
//              case 2: w_shiftOut = ({32{w_shiftIn[31]}}<<(32-w_shiftAmt))|(w_shiftIn>>w_shiftAmt); break;
//              Arithmetic Shift Right should be modified.
//              case 2: w_shiftOut = ({w_shiftIn[32]{w_shiftIn[31]}}<<(32-w_shiftAmt))|(w_shiftIn>>w_shiftAmt);
break;
                case 2: w_shiftOut = w_shiftIn >> w_shiftAmt; break;
                case 3: w_shiftOut = (w_shiftIn >> w_shiftAmt) | (w_shiftIn << (32-w_shiftAmt)); break;
                        //Rotate
                default:
                 break;
        }
        shiftOut.write(w_shiftOut);
}
```

```
/*
 * Datapath: Data-path architecture
 * for ICS(Intelligent Configurable Switch)RISC Core(header file for datapath)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: datapath.h
 * Revision history: Version1
 * Date: 30/4/2005
 */

#include "systemc.h"
#include "pc.h"
#include "sr.h"
#include "lf.h"
#include "regFile.h"
#include "aluDef.h"
#include "aluICS.h"
#include "shifter.h"
#include "mul.h"
```

```
SC_MODULE(datapath) {
        sc_in<bool>             clock;
        sc_in<bool>             reset;
        sc_in<sc_uint<32> >     ir_m;       //Immediate Data
        sc_in<bool>             cmpFlag;    //Compare Flag
        sc_in<bool>             srOEn;      //Status Register Enable
        sc_in<bool>             srWbEn;     //Status Register Read/Write Enable
        sc_in<sc_uint<4> >      aluCtl;     //ALU Control Signal
        sc_in<bool>             aluOEn;     //ALU Output Enable
        sc_in<sc_uint<3> >      shiftCtl;   //Shifter Control Signal
        sc_in<bool>             shiftOEn;   //Shifter Output Enable
        sc_in<bool>             mulOEn;     //Multiplier Output Enable
        sc_in<sc_uint<5> >      opA1dx;     //Operand A Index
        sc_in<sc_uint<5> >      opB1dx;     //Operand B Index
        sc_in<bool>             rdAOEn;     //Read A Output Enable
        sc_in<bool>             rdBOEn;     //Read B Output Enable
        sc_in<sc_uint<5> >      wb1dx;      //Writeback Index
        sc_in<bool>             wbEn;       //Writeback Enable
        sc_in<bool>             immOEn;     //Immediate Output Enable
        sc_in<bool>             iAregCtl;   //Instruction Address Register Control
        sc_in<boo!>             dAregCtl;   //Data Address Register Control
        sc_in<bool>             dInCtl;     //Data Input Control
        sc_in<bool>             dOutCtl;    //Data Output Control
        sc_in<sc_uint<32> >     dIn;        //Data Input
        sc_in<bool>             lbEn;       //Loop Buffer Enable
        sc_in<bool>             lbRWEn;     //Loop Buffer Read/Write Enable

        //Output Signals
        sc_out<bool>            zFlag;      //Zero Flag
        sc_out<sc_uint<32> >    iAddr;      //Instruction Address
        sc_out<sc_uint<32> >    dAddr;      //Data Address
        sc_out<sc_uint<32> >    dOut;       //Data Output

        //Temp Signals
        sc_signal<sc_uint<32> > busA, busB, busW;
        sc_signal<sc_uint<32> > s_aluOut;   //ALU Output Signal
        sc_signal<sc_uint<4> >  s_condFlag;         //Conditional Flag
        sc_signal<sc_uint<32> > s_shiftOut;         //Shifter Output Signal
        sc_signal<sc_uint<32> > s_mulOut;           //Multiplier Output Signal
        sc_signal<sc_uint<4> >  tmpbusA, tmpbusW;   //Temp Signals for SR
        sc_signal<sc_uint<5> >  s_shiftAmt;         //Temp Signal for Shifter
        sc_signal<sc_uint<32> > pIAddr;             //Instruction Address from PC
        sc_signal<sc_uint<32> > lIAddr;             //Instruction Address from LF

        void do_outCtl();           //Function for Output Control
        void do_inOutReg();         //Function for Data In/Out Register
        void do_sigDiv() {          //Function for Gen. Signals for Status Register
        sc_uint<32> busA1, busW1, busB1;
        busA1 = busA;
        busW1 = busW;
        busB1 = busB;

        tmpbusA = busA1.range(31,28);
        tmpbusW = busW1.range(31,28);
        s_shiftAmt = busB1.range(15,11);  //Signal for Shift Amount Control
        }

        pc*         ipc;
        sr*         isr;
        lf*         ilf;
        regFile*    iregFile;
        aluICS*     ialuICS;
        shifter*    ishifter;
        mul*        imul;

        SC_CTOR(datapath) {
                ipc=new pc("pc");
                ipc->clock(clock);      ipc->reset(reset);              ipc->lAregCtl(lAregCtl);
                ipc->dAregCtl(dAregCtl);        ipc->aluOut(s_aluOut);
```

```
                ipc->iAddr(pIAddr);             ipc->dAddr(dAddr);
                isr=new sr("sr");
                isr->clock(clock);              isr->reset(reset);      isr->condFlag(s_condFlag);
                isr->wbData(tmpbusW);           isr->wbSel(cmpFlag);    isr->srOEn(srOEn);
                isr->srWbEn(srWbEn);            isr->zFlag(zFlag);      isr->rdData(tmpbusA);
                ialuICS=new aluICS("aluICS");
                ialuICS->aluAIn(busA);          ialuICS->aluBIn(busB);          ialuICS->aluCtl(aluCtl);
                ialuICS->aluOut(s_aluOut);      ialuICS->condFlag(s_condFlag);
                ishifter=new shifter("shifter");
                ishifter->shiftIn(busA);ishifter->shiftAmt(s_shiftAmt);
                ishifter->shiftCtl(shiftCtl);   ishifter->shiftOut(s_shiftOut);
                imul=new mul("mul");
                imul->clock(clock);             imul->reset(reset);             imul->mulAIn(busA);
                imul->mulBIn(busB);             imul->mulOut(s_mulOut);
                ilf=new lf("lf");
                ilf->clock(clock);              ilf->reset(reset);              ilf->lbRWEn(lbRWEn);
                ilf->lbEn(lbEn);                ilf->lAddrIn(pIAddr);           ilf->lAddrOut(lIAddr);
                iregFile=new regFile("regFile");
                iregFile->clock(clock);         iregFile->rdAIdx(opAIdx);       iregFile->rdBIdx(opBIdx);
                iregFile->rdAOEn(rdAOEn);       iregFile->rdBOEn(rdBOEn);       iregFile->wbIdx(wbIdx);
                iregFile->wbData(busW);         iregFile->wbEn(wbEn);           iregFile->pc(iAddr);
                iregFile->rdAData(busA);        iregFile->rdBData(busB);

                SC_METHOD(do_outCtl);
                sensitive << immOEn << aluOEn << shiftOEn << mulOEn << lbEn << clock << reset;
                SC_METHOD(do_inOutReg);
                sensitive << dInCtl << dOutCtl;
                SC_METHOD(do_sigDiv);
                sensitive << clock << reset;

#ifdef SIM
                zFlag.initialize(0);
                iAddr.initialize(0);
                dAddr.initialize(0);
                dOut.initialize(0);
#endif
        }
};

/*
 * Datapath: Data-path architecture
 * for ICS(Intelligent Configurable Switch)RISC Core(Source file for datapath)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: datapath.cpp
 * Revision history: Version1
 * Date: 30/4/2005
 */

#include "datapath.h"

void datapath::do_outCtl() {
        if (immOEn) {
                busB = imm;
        }
        if (aluOEn) {
                busW = s_aluOut;
        }
        if (shiftOEn) {
                busW = s_shiftOut;
        }
        if (mulOEn) {
                busW = s_mulOut;
        }
        if (lbEn) {                             //Loop Buffer Addressing
                iAddr = lIAddr;                 //Instruction Address from LB
        } else {
                iAddr = pIAddr;                 //Instruction Address from PC
        }
}
```

```
void datapath::do_InOutReg() {
        if (dInCtl) {                           //Data Input Register
                busW = dIn;
        }
        if (dOutCtl) {                          //Data Output Register
                dOut = busB;
        }
}
```

3.2 Control Architecture

```
/*
 * Def: Macros for ICS_RISC
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: def.h
 * Revision history: Version1
 * Date: 2/5/2005
 */

#define INST_ALUIS      0       //ALU Imm. Short(1 Inst. word)
#define INST_ALUIL      1       //ALU Imm. Long(2 Inst. word)
#define INST_ALUR       2       //ALU Register
#define INST_ALULB      3       //ALU Loop Buffer Addressing
#define INST_SHRO       4       //Shift/Rotate
#define INST_LOAD       5       //Load
#define INST_STORE      6       //Store
#define INST_BRANCH     7       //Branch
#define INST_PECON      8       //PE Control
#define INST_DMA        9       //DMA Control
#define INST_MUL        10      //Multiply

#define COND_EQ         0       //Equal
#define COND_NE         1       //Not Equal
#define COND_AL         2       //Always
//#define COND_NV       3       //Never

#define OP_MOVA         0
#define OP_MOVB         1
#define OP_AND          2
#define OP_OR           3
#define OP_XOR          4
#define OP_NOT          5
#define OP_ADD          6
#define OP_SUB          7
#define OP_CMP          8
#define OP_MSR          9
#define OP_MRS          10

#define SH_LSL          0
#define SH_LSR          1
#define SH_ASR          2
#define SH_ROT          3
```

```
/*
 * Fetch: Fetch Unit for ICS_RISC(header file for fetch)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: fetch.h
 * Revision history: Version1
 * Date: 1/5/2005
 */
```

```
#include "systemc.h"

SC_MODULE(fetch) {
        sc_in<bool>             clock;
        sc_in<sc_uint<32> >     dIn;        //Instruction Data
        sc_out<sc_uint<32> >    fInst;      //Fetched Data

        void do_fetch();

        SC_CTOR(fetch) {
                SC_METHOD(do_fetch);
                sensitive << clock.pos() << dIn;
#ifdef SIM
                fInst.initialize(0);
#endif
        }
};

/*
 * Fetch: Fetch Unit for ICS_RISC(Source file for fetch)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: fetch.cpp
 * Revision history: Version1
 * Date: 1/5/2005
 */

#include "fetch.h"

void fetch::do_fetch() {
        fInst = dIn.read();
}
```

```
/*
 * Decode: Instruction Decoder Unit for ICS_RISC(header file for decode)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: decode.h
 * Revision history: Version1
 * Date: 1/5/2005
 */

#include "systemc.h"
#include "def.h"

SC_MODULE(decode) {
        sc_in<bool>             clock;
        sc_in<bool>             reset;
        sc_in<sc_uint<32> >     fInst;      //Fetched Instruction
        sc_in<bool>             flush;      //Pipeline Flush

        sc_out<bool>            refill;     //Pipeline Refill
        sc_out<sc_uint<4> >     instId;     //Instruction ID
        sc_out<sc_uint<3> >     cond;       //Branch condition
        sc_out<sc_uint<4> >     opcode;     //Op code
        sc_out<sc_uint<3> >     shift;      //shift control signal
        sc_out<sc_uint<5> >     rs1Idx;     //Rb/Rs1 Index
        sc_out<sc_uint<5> >     rs2Idx;     //Rs2 Index
        sc_out<sc_uint<5> >     rdIdx;      //Rd Index
        sc_out<sc_uint<32> >    imm;        //Immediate data
        sc_out<bool>            immFlag;    //Immediate operand flag
        sc_out<bool>            cmpFlag;    //Compare flag(update status register/no writeback)
        sc_out<bool>            branchFlag; //Branch Flag
        sc_out<bool>            exitFlag;   //End of simulaion flag
        sc_out<bool>            srOEn;      //Status register output enable
        sc_out<bool>            srWbEn;     //Status register read/write enable
        //Extended Output
```

```
        //For Loop Buffer
        sc_out<bool>                lbEn;               //Loop Buffer Enable
        sc_out<bool>                lbRWEn;             //Loop Buffer Read/Write Enable
        //PE Control
        sc_out<sc_uint<3> >         PEOp;               //PE Execute Operations
        sc_out<sc_uint<2> >         PEOpmode;           //PE Operation Mode Selection
        sc_out<sc_uint<2> >         PEConfig;           //PE Configuration
        sc_out<sc_uint<4> >         PESel;              //PE Sel
        //DMA Control
        sc_out<sc_uint<3> >         DMAOp;              //DMA Execute Operations Selection
        sc_out<bool>                DFBSel;             //Data Frame Buffer Selection(2 Sets)
        sc_out<sc_uint<4> >         dataAmt;            //Amount of Data to Transfer
        sc_out<sc_uint<6> >         startAddrDFB;       //Start Address of DFB(Source/Dest.)
        sc_out<bool>                SRAMRegSel;         //Select between SRAM/ICS_RISC Reg(Source/Dest.)
        sc_out<sc_uint<5> >         startAddrSRAMReg; //Start Address of SRAM/ICS_RISC Reg(S/D)
        sc_out<bool>                memSel;             //Memory Selection(Program/Data)
        sc_out<sc_uint<10> >  startAddrProgDaMem; //Start Address of Program/Data Memory(S/D)

        sc_uint<7>                  fInstId;            //fInst[31:25] for extract Instruction ID
        sc_uint<4>                  instIdTmp;          //Describe the 10 sets of Instruction ID

        void do_pipelineCtl();      //Function for pipeline control
        void do_instId();           //Function for extract instruction ID
        void do_cond();             //Function for condition
        void do_fieldExt();         //Function for instruction field extraction

        SC_CTOR(decode) {
                SC_METHOD(do_pipelineCtl);
                sensitive << clock.pos() << flush;
                SC_METHOD(do_instId);
                sensitive << clock.pos() << fInst;
                SC_METHOD(do_cond);
                sensitive << clock.pos();
                SC_METHOD(do_fieldExt);
                sensitive << clock.pos() << fInst;
        }
};

/*
 * Decode: Instruction Decoder Unit for ICS_RISC(source file for decode)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: decode.cpp
 * Revision history: Version1
 * Date: 2/5/2005
 */

#include "decode.h"

void decode::do_pipelineCtl() {             //Function for Pipeline Control
        bool    refillTmp;
        if (reset) {
                refillTmp = 1;
        } else {
                if (flush) {
                        refillTmp = 1;
                        refill = 1;
                } else {
                        refillTmp = 0;
                        refill = refillTmp;
                }
        }
}
```

```
/*
 * Execute: Execute Unit for ICS_RISC(header file for execute)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
```

```
 * File name: execute.h
 * Revision history: Version1
 * Date: 2/5/2005
 */

#include "systemc.h"
#include "def.h"

//Opernad A Control
#define ARD             0;              //Operand A : Rd
#define ARS1            1;              //Operand A : Rs1
#define APC             2;              //Operand A : PC
//Operand B Control
#define BIM             0;              //Operand B : Immediate
#define BRS2            1;              //Operand B : Rs2/ShiftAmt
#define BRD             2;              //Operand B : Rd


SC_MODULE(execute) {
        sc_in<bool>         clock;
        sc_in<sc_uint<4> >  instId;     //Instruction ID
        sc_in<sc_uint<3> >  cond;       //Condition
        sc_in<sc_uint<4> >  opcode;     //Op code
        sc_in<sc_uint<3> >  shift;      //Shift Type
        sc_in<sc_uint<5> >  rs1Idx;     //Rs1 Index
        sc_in<sc_uint<5> >  rs2Idx;     //Rb/Rs2 Index
        sc_in<sc_uint<5> >  rdIdx;      //Rd Index
        sc_in<sc_uint<32> > imm;        //Immediate data
        sc_in<bool>         immFlag;    //Immediate Operand Flag
        sc_in<bool>         cmpFlag;    //Compare Flag (update SR, No writeback)
        sc_in<bool>         srOEn;      //Status Register Output Enable
        sc_in<bool>         srWbEn;     //Status Register Writeback Enable
        //Output Signals
        sc_out<sc_uint<4> > aluCtl;     //ALU Control
        sc_out<bool>        aluOEn;     //ALU Output Enable
        sc_out<sc_uint<3> > shiftCtl;   //Shifter Control
        sc_out<bool>        shiftOEn;   //Shifter Output Enable
        sc_out<bool>        mulOEn;     //Multiplier Output Enable
        sc_out<sc_uint<5> > opAIdx;     //Operand A Index
        sc_out<sc_uint<5> > opBIdx;     //Operand B Index/Shift Amt
        sc_out<bool>        rdAOEn;     //Read A Output Enable
        sc_out<bool>        rdBOEn;     //Read B Output Enable
        sc_out<sc_uint<5> > wbIdx;      //Writeback Index
        sc_out<bool>        wbEn;       //Writeback Enable
        sc_out<bool>        immOEn;     //Immediate Output Enable
        sc_out<bool>        iAregCtl;   //Instruction Address Register Control
        sc_out<bool>        dAregCtl;   //Data Address Register Control
        sc_out<bool>        dInCtl;     //Data Input Control
        sc_out<bool>        dOutCtl;    //Data Output Control
//      sc_out<sc_uint<5> > shiftAmt;   //Shift Amount

        void do_ctlSigGen();  //Function for Control Signal Generate
        void do_opSel();      //Function for Select Input OperandA,B
        void do_aluCtl();     //Function for Arrange AluCtl Signals
        void do_shiftCtl();   //Function for Arrange ShiftCtl Signals

        sc_uint<4> opcodeTmp;
        sc_uint<2> opA, opB;

        SC_CTOR(execute) {
                SC_METHOD(do_ctlSigGen);
                sensitive << clock.pos() << instId;
                SC_METHOD(do_opSel);
                sensitive << clock.pos() << rdIdx << rs1Idx << rs2Idx;
                SC_METHOD(do_aluCtl);
                sensitive << clock.pos() << opcode;
                SC_METHOD(do_shiftCtl);
                sensitive << clock.pos() << shift;

#ifdef SIM
```

```
                    aluCtl.initialize(0);
                    aluOEn.initialize(0);
                    shiftCtl.initialize(0);
                    shiftOEn.initialize(0);
                    mulOEn.initialize(0);
                    opAIdx.initialize(0);
                    opBIdx.initialize(0);
                    roAOEn.initialize(0);
                    rdBOEn.initialize(0);
                    wbIdx.initialize(0);
                    immOEn.initialize(0);
                    iAregCtl.initialize(0);
                    dAregCtl.initialize(0);
                    dInCtl.initialize(0);
                    dOutCtl.initialize(0);
#endif
          }
};

/*
 * Execute: Execute Unit for ICS_RISC(source file for execute)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: execute.cpp
 * Revision history: Version1
 * Date: 4/5/2005
 */

#include "execute.h"

void execute::do_ctlSigGen() {
          sc_uint<4>   topcodeTmp;
          sc_uint<2> topA, topB;
          bool         taluOEn, tshiftOEn, tmulOEn, twbEn, tiAregCtl, tdAregCtl, tdInCtl, tdOutCtl;
          bool         wbEnTmp;
          sc_uint<4> instIdTmp;
          instIdTmp = instId.read();
//        sc_uint<5> tshiftAmt;

          if (instIdTmp == 0) {   //INST_ALUIS
                    topcodeTmp          = opcode.read();
                    taluOEn             = 1;
                    tshiftOEn           = 0;
                    tmulOEn             = 0;
                    topA                = ARD;
                    topB                = BIM;
                    twbEn               = 1;
                    tiAregCtl           = 0;
                    tdAregCtl           = 0;
                    tdInCtl             = 0;
                    tdOutCtl            = 0;
          } else if (instIdTmp == 1) {        //INST_ALUIL
                    topcodeTmp          = opcode.read();
                    taluOEn             = 1;
                    tshiftOEn           = 0;
                    tmulOEn             = 0;
                    topA                = ARD;
                    topB                = BIM;
                    twbEn               = 1;
                    tiAregCtl           = 0;
                    tdAregCtl           = 0;
                    tdInCtl             = 0;
                    tdOutCtl            = 0;
          } else if (instIdTmp == 2) {        //INST_ALUR
                    topcodeTmp          = opcode.read();
                    taluOEn             = 1;
                    tshiftOEn           = 0;
                    tmulOEn             = 0;
                    topA                = ARS1;
                    topB                = BRS2;
```

```
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        } else if (instIdTmp == 3) {        //INST_ALULB
                topcodeTmp      = opcode.read();
                taluOEn         = 1;
                tshiftOEn       = 0;
                tmulOEn         = 0;
                topA            = ARS1;
                topB            = BRS2;
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        } else if (instIdTmp == 4) {        //INST_SHRO
                topcodeTmp      = 0;
                taluOEn         = 0;
                tshiftOEn       = 1;
                tmulOEn         = 0;
                topA            = ARS1;
                topB            = BRS2;
//              tshiftAmt       = BRS2;     //BRS2 = ShiftAmt
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        } else if (instIdTmp == 5) {        //INST_LOAD
                topcodeTmp      = OP_MOVA;
                taluOEn         = 0;
                tshiftOEn       = 0;
                tmulOEn         = 0;
                topA            = ARS1;
                topB            = BRS2;
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 1;
                tdInCtl         = 1;
                tdOutCtl        = 0;
        } else if (instIdTmp == 6) {        //INST_STORE
                topcodeTmp      = OP_MOVA;
                taluOEn         = 0;
                tshiftOEn       = 0;
                tmulOEn         = 0;
                topA            = ARS1;
                topB            = BRD;
                twbEn           = 0;
                tiAregCtl       = 0;
                tdAregCtl       = 1;
                tdInCtl         = 0;
                tdOutCtl        = 1;
        } else if (instIdTmp == 7) {        //INST_BRANCH
                topcodeTmp      = OP_ADD;
                taluOEn         = 1;
                tshiftOEn       = 0;
                tmulOEn         = 0;
                topA            = APC;
                topB            = BIM;
                twbEn           = 0;
                tiAregCtl       = 1;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        } else if (instIdTmp == 10) {       //INST_MUL
                topcodeTmp      = 0;
                taluOEn         = 0;
                tshiftOEn       = 0;
```

```
                tmulOEn         = 1;
                topA            = ARS1;
                topB            = BRS2;
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        } else if (instIdTmp == 8) {    //INST_PECON--Should be modified
                topcodeTmp      = opcode.read();
                taluOEn         = 1;
                tshiftOEn       = 0;
                tmulOEn         = 0;
                topA            = ARS1;
                topB            = BRS2;
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        } else if (instIdTmp == 9) {    //INST_DMA--Should be modified
                topcodeTmp      = opcode.read();
                taluOEn         = 1;
                tshiftOEn       = 0;
                tmulOEn         = 0;
                topA            = ARS1;
                topB            = BRS2;
                twbEn           = 1;
                tiAregCtl       = 0;
                tdAregCtl       = 0;
                tdInCtl         = 0;
                tdOutCtl        = 0;
        }

        opcodeTmp       = topcodeTmp;
        aluOEn          = taluOEn;
        shiftOEn        = tshiftOEn;
        mulOEn          = tmulOEn;
        opA             = topA;
        opB             = topB;
        wbEnTmp         = twbEn;
        iAregCtl        = tiAregCtl;
        dAregCtl        = tdAregCtl;
        dInCtl          = tdInCtl;
        dOutCtl         = tdOutCtl;
//      shiftAmt        = tshiftAmt;

        rdAOEn = ~srOEn;

        if (opB != 0) {
                rdBOEn = 1;
        }
        if ((cmpFlag==0) && (wbEnTmp==1)) {
                wbEn = 1;
        }
        if (opB == 0) {
                immOEn = 1;
        }
}

void execute::do_opSel() {
        if (opA == 0) {                         //ARD
                opAIdx = rdIdx;
        } else if (opA == 1) {  //ARS1
                opAIdx = rs1Idx;
        } else if (opA == 2) {  //APC
                opAIdx = 15;
        }

        if (opB == 0) {                         //BIM
```

```
                opBIdx = 3;
        } else if (opB == 1) {      //BRS2
                opBIdx = rs2Idx;
        } else if (opB == 2) {      //BRD
                opBIdx = rdIdx;
        }
        wbIdx = rdIdx;
}

void execute::do_aluCtl() {
        opcodeTmp = opcode.read();

        switch(opcodeTmp) {
                case (OP_MOVA)      :       aluCtl = 0;         break;
                case (OP_MOVB)      :       aluCtl = 1;         break;
                case (OP_AND)       :       aluCtl = 2;         break;
                case (OP_OR)        :       aluCtl = 3;         break;
                case (OP_XOR)       :       aluCtl = 4;         break;
                case (OP_NOT)       :       aluCtl = 5;         break;
                case (OP_ADD)       :       aluCtl = 6;         break;
                case (OP_SUB)       :       aluCtl = 7;         break;
                case (OP_CMP)       :       aluCtl = 8;         break;
                case (OP_MSR)       :       aluCtl = 0;         break;
                case (OP_MRS)       :       aluCtl = 0;         break;
                default     :                                   break;
        }
}

void execute::do_shiftCtl() {
        sc_uint<3> shiftTmp;
        shiftTmp = shift.read();

        switch(shiftTmp) {
                case SH_LSL         :       shiftCtl = 0;       break;
                case SH_LSR         :       shiftCtl = 1;       break;
                case SH_ASR         :       shiftCtl = 2;       break;
                case SH_ROT         :       shiftCtl = 3;       break;
                default             :                           break;
        }
}
```

```
/*
 * Control: Control Arch. for ICS_RISC(header file for control)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(cklm@student.ecu.edu.au)
 * File name: control.h
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "systemc.h"
#include "def.h"
#include "fetch.h"
#include "decode.h"
#include "execute.h"
#include "debug.h"

SC_MODULE(control) {
        sc_in<bool>             clock;          //Clock
        sc_in<bool>             reset;
        sc_in<sc_uint<32> >     dIn;            //Data Input
        sc_in<bool>             zFlag;          //Zero Flag

        sc_out<sc_uint<32> >    imm;            //Immediate Data
        sc_out<bool>            cmpFlag;    //Compare Flag
        sc_out<bool>            srOEn;          //SR Output Enable
        sc_out<bool>            srWbEn;         //SR Writeback Enable
        sc_out<sc_uint<4> >     aluCtl;         //ALU Control
```

```
        sc_out<bool>                aluOEn;             //ALU Output Enable
        sc_out<sc_uint<3> >         shiftCtl;   //Shifter Control
        sc_out<bool>                shiftOEn;   //Shifter Output Enable
        sc_out<bool>                mulOEn;             //Multiplier Output Enable
        sc_out<sc_uint<5> >         opAIdx;             //Operand A Index
        sc_out<sc_uint<5> >         opBIdx;             //Operand B Index
        sc_out<bool>                rdAOEn;             //Read A Output Enable
        sc_out<bool>                rdBOEn;             //Read B Output Enable
        sc_out<sc_uint<5> >         wbIdx;              //Writeback Index
        sc_out<bool>                wbEn;               //Writeback Enable
        sc_out<bool>                immOEn;             //Immediate Output Enable
        sc_out<bool>                iAregCtl;   //Instruction Address Register Control
        sc_out<bool>                dAregCtl;   //Data Address Register Control
        sc_out<bool>                dInCtl;             //Data Input Control
        sc_out<bool>                dOutCtl;    //DAta Output Control
        //Extended Output
//      sc_out<sc_uint<5> >         shiftAmt;   //Shift Amount
        sc_out<bool>                lbEn;               //Loop Buffer Enable
        sc_out<bool>                lbRWEn;             //Loop Buffer Read/Write Enable
        sc_out<sc_uint<3> >         PEOp;               //PE Execution Operation
        sc_out<sc_uint<2> >         PEOpmode;           //PE Operation Mode Selection
        sc_out<sc_uint<2> >         PEConfig;   //PE Configuration
        sc_out<sc_uint<4> >         PESel;              //PE Selection
        sc_out<sc_uint<3> >         DMAOp;              //DMA Operation Selection
        sc_out<bool>                DFBSel;             //Data Frame Buffer Selection
        sc_out<sc_uint<4> >         dataAmt;    //Amount of Data to Transfer
        sc_out<sc_uint<6> >         startAddrDFB;       //Start Address of DFB(Source/Dest.)
        sc_out<bool>                SRAMRegSel;         //Select between SRAM/ICS_RISC Reg(S/D)
        sc_out<sc_uint<5> >         startAddrSRAMReg; //Start Address of SRAM/ICS_RISC Reg(S/D)
        sc_out<bool>                memSel;             //Memory Selection(Program/Data)
        sc_out<sc_uint<10> >  startAddrProgDaMem;       //Start Address of Program/Data Memory(S/D)

        sc_signal<sc_uint<32> >     finst;      //Fetched Instruction Data
        sc_signal<bool>             flush;      //Pipeline Flush
        sc_signal<bool>             refill;     //Pipeline Refill
        sc_signal<sc_uint<4> >      dInstId;    //Instruction ID
        sc_signal<sc_uint<3> >      dCond;      //Condition
        sc_signal<sc_uint<4> >      opcode;     //Opcode
        sc_signal<sc_uint<3> >      shift;      //Shift Control
        sc_signal<sc_uint<5> >      rs1Idx;     //Rs1 Index
        sc_signal<sc_uint<5> >      rs2Idx;     //Rs2 Index
        sc_signal<sc_uint<5> >      rdIdx;      //Rd Index
        sc_signal<sc_uint<32> >     dImm;       //Immediate Data
        sc_signal<bool>             immFlag;    //Immediate Operand Flag
        sc_signal<bool>             dCmpFlag;//Compare Flag
        sc_signal<bool>             dBranchFlag;//Branch Flag
        sc_signal<bool>             dExitFlag; //End of Simulation Flag
        sc_signal<bool>             dSrOEn, dSrWbEn; //SR Read/Write Enable
        sc_signal<bool>             dLbEn, dLbRWEn;     //LB Enable Read/Write Enable
        sc_signal<sc_uint<3> >      dPEOp;              //PE Execution Operation
        sc_signal<sc_uint<2> >      dPEOpmode;          //PE Operation Mode Selection
        sc_signal<sc_uint<2> >      dPEConfig;          //PE Configuration
        sc_signal<sc_uint<4> >      dPESel;             //PE Selection
        sc_signal<sc_uint<3> >      dDMAOp;             //DMA Operation Selection
        sc_signal<bool>             dDFBSel;    //DFB Selection
        sc_signal<sc_uint<4> >      dDataAmt;//Amount of Data to Transfer
        sc_signal<sc_uint<6> >      dStartAddrDFB;
        sc_signal<bool>             dSRAMRegSel;
        sc_signal<sc_uint<5> >      dStartAddrSRAMReg;
        sc_signal<bool>             dMemSel;
        sc_signal<sc_uint<10> >     dStartAddrProgDaMem;

        sc_signal<sc_uint<4> >      dAluCtl;    //ALU Control
        sc_signal<bool>             dAluOEn;    //ALU Output Enable
        sc_signal<sc_uint<3> >      dShiftCtl;  //Shift Control
        sc_signal<bool>             dShiftOEn;//Shift Output Enable
        sc_signal<bool>             dMulOEn; //Multiplier Output Enable
        sc_signal<sc_uint<5> >      dOpAIdx;  //Operand A Index
        sc_signal<sc_uint<5> >      dOpBIdx;  //Operand B Index
        sc_signal<bool>             dRdAOEn;//Read A Output Enable
```

```
        sc_signal<bool>             dRdBOEn;//Read B Output Enable
        sc_signal<sc_uint<5> >      dWbIdx;   //Writeback Index
        sc_signal<bool>             dWbEn;              //Writeback Enable
        sc_signal<bool>             dImmOEn;//Immediate Output Enable
        sc_signal<bool>             dIAregCtl; //Instruction Address Register Control
        sc_signal<bool>             dDAregCtl;          //Data Address Register Control
        sc_signal<bool>             dDInCtl;   //Data Input Control
        sc_signal<bool>             dDOutCtl; //Data Output Control
//      sc_signal<sc_uint<5> >      dShiftAmt; //Shift Amount

        sc_signal<int>              instIdText; //Instruction ID Debug Information
        sc_signal<int>              aluText;    //ALU Debug Information

        //Pipeline Registers
        sc_uint<4>                  instId;             //Instruction ID
        sc_uint<3>                  cond;               //Execution Condition
//      bool                        cmpFlag;            //Compare Flag
        bool                        branchFlag;          //Branch Flag
        bool                        eExitFlag;          //Exit Flag
//      bool                        srOEn;              //SR Output Enable
        bool                        eSrWbEn;            //SR Writeback Enable
//      sc_uint<4>                  aluCtl;             //ALU Control
//      bool                        aluOEn;             //ALU Output Enable
//      sc_uint<3>                  shiftCtl;           //Shifter Output Control
//      bool                        shiftOEn;           //Shifter Output Enable
//      bool                        mulOEn;             //Multiplier Output Enable
//      sc_uint<5>                  opAIdx;             //Operand A Index
//      sc_uint<5>                  opBIdx;             //Operand B Index
//      bool                        rdAOEn;             //Read A Output Enable
//      bool                        rdBOEn;             //Read B Output Enable
//      sc_uint<5>                  wbIdx;              //Writeback Index
        bool                        eWbEn;              //Writeback Enable
//      sc_uint<32>                 imm;                //Immediate Data
//      bool                        immOEn;             //Immediate Output Enable
        bool                        eIAregCtl;          //Instruction Address Register Control
//      bool                        dAregCtl;           //Data Address Register Control
//      bool                        dInCtl;             //Data Input Control
        bool                        eDOutCtl;           //Data Output Control

        void do_pipeReg();
        void do_condExe();

        fetch*      ifetch;
        decode*     idecode;
        execute*    iexecute;
        debug*      idebug;

        SC_CTOR(control) {
                ifetch=new fetch("fetch");
                ifetch->clock(clock);   ifetch->dIn(dIn);       ifetch->fInst(fInst);
                idecode=new decode("decode");
                idecode->clock(clock);          idecode->reset(reset);  idecode->fInst(fInst);
                idecode->flush(flush);          idecode->refill(refill);  idecode->instId(dInstId);
                idecode->cond(dCond);           idecode->opcode(opcode);        idecode->shift(shift);
                idecode->rs1Idx(rs1Idx);        idecode->rs2Idx(rs2Idx);        idecode->rdIdx(rdIdx);
                idecode->imm(dImm);             idecode->immFlag(immFlag);      idecode-
>cmpFlag(dCmpFlag);
                idecode->branchFlag(dBranchFlag); idecode->exitFlag(dExitFlag); idecode->srOEn(dSrOEn);
                idecode->srWbEn(dSrWbEn);       idecode->lbEn(dLbEn);                   idecode-
>lbRWEn(dLbRWEn);
                idecode->PEOp(dPEOp);           idecode->PEOpmode(dPEOpmode);           idecode-
>PEConfig(dPEConfig);
                idecode->PESel(dPESel);         idecode->DMAOp(dDMAOp);                 idecode-
>DFBSel(dDFBSel);
                idecode->dataAmt(dDataAmt);     idecode->startAddrDFB(dStartAddrDFB);
                idecode->SRAMRegSel(dSRAMRegSel); idecode->startAddrSRAMReg(dStartAddrSRAMReg);
                idecode->memSel(dMemSel);       idecode->startAddrProgDaMem(dStartAddrProgDaMem);
                iexecute=new execute("execute");
                iexecute->clock(clock);         iexecute->instId(dInstId);      iexecute->cond(dCond);
                iexecute->opcode(opcode);       iexecute->shift(shift);  iexecute->rs1Idx(rs1Idx);
```

```
                    iexecute->rs2Idx(rs2Idx);       iexecute->rdIdx(rdIdx);         iexecute->imm(dImm);
                    iexecute->immFlag(immFlag);     iexecute->cmpFlag(dCmpFlag); iexecute->srOEn(dSrOEn);
                    iexecute->srWbEn(dSrWbEn);      iexecute->aluCtl(dAluCtl);      iexecute->aluOEn(dAluOEn);
                    iexecute->shiftCtl(dShiftCtl);  iexecute->shiftOEn(dShiftOEn); iexecute->mulOEn(dMulOEn);
                    iexecute->opAIdx(dOpAIdx);      iexecute->opBIdx(dOpBIdx);      iexecute-
>rdAOEn(dRdAOEn);
                    iexecute->rdBOEn(dRdBOEn);      iexecute->wbIdx(dWbIdx);        iexecute->wbEn(dWbEn);
                    iexecute->immOEn(dImmOEn);      iexecute->iAregCtl(dIAregCtl);  iexecute-
>dAregCtl(dDAregCtl);
                    iexecute->dInCtl(dDInCtl);      iexecute->dOutCtl(dDOutCtl);    //iexecute-
>shiftAmt(dShiftAmt);
                    idebug=new debug("debug");
                    idebug->instId(dInstId);        idebug->aluCtl(dAluCtl);        idebug-
>instIdText(instIdText);
                    idebug->aluText(aluText);

                    SC_METHOD(do_pipeReg);
                    sensitive << clock.pos() << reset;
                    SC_METHOD(do_condExe);
                    sensitive << clock.pos();
#ifdef SIM
                    imm.initialize(0);
                    cmpFlag.initialize(0);
                    srOEn.initialize(0);
                    srWbEn.initialize(0);
                    aluCtl.initialize(0);
                    aluOEn.initialize(0);
                    shiftCtl.initialize(0);
                    shiftOEn.initialize(0);
                    mulOEn.initialize(0);
                    opAIdx.initialize(0);
                    opBIdx.initialize(0);
                    rdAOEn.initialize(0);
                    rdBOEn.initialize(0);
                    wbIdx.initialize(0);
                    wbEn.initialize(0);
                    immOEn.initialize(0);
                    iAregCtl.initialize(0);
                    dAregCtl.initialize(0);
                    dInCtl.initialize(0);
                    dOutCtl.initialize(0);
#endif
          }
};

/*
 * Control: Control Arch. for ICS_RISC(source file for control)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: control.cpp
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "control.h"

void control::do_pipeReg() {
          if (reset) {
                    cond = 0;
                    eSrWbEn = 0;
                    eWbEn = 0;
                    eIAregCtl = 0;
                    branchFlag = 0;
                    eExitFlag = 0;
          } else {
                    instId      = dInstId;
                    cond        = dCond;
                    cmpFlag     = dCmpFlag;
                    branchFlag= dBranchFlag;
                    eExitFlag   = dExitFlag;
```

```
                srOEn     = dSrOEn;
                eSrWbEn   = dSrWbEn;
                aluCtl    = dAluCtl;
                aluOEn    = dAluOEn;
                shiftCtl  = dShiftCtl;
                shiftOEn  = dShiftOEn;
                mulOEn    = dMulOEn;
                opAIdx    = dOpAIdx;
                opBIdx    = dOpBIdx;
                rdAOEn    = dRdAOEn;
                rdBOEn    = dRdBOEn;
                wbIdx     = dWbIdx;
                eWbEn     = dWbEn;
                immOEn    = dImmOEn;
                imm       = dImm;
                eIAregCtl = dIAregCtl;
                dAregCtl  = dDAregCtl;
                dInCtl    = dDInCtl;
                eDOutCtl  = dDOutCtl;
                lbEn      = dLbEn;
                lbRWEn    = dLbRWEn;
                PEOp      = dPEOp;
                PEOpmode= dPEOpmode;
                PEConfig  = dPEConfig;
                PESel     = dPESel;
                DMAOp     = dDMAOp;
                DFBSel    = dDFBSel;
                dataAmt   = dDataAmt;
                startAddrDFB =dStartAddrDFB;
                SRAMRegSel=dSRAMRegSel;
                startAddrSRAMReg = dStartAddrSRAMReg;
                memSel    = dMemSel;
                startAddrProgDaMem = dStartAddrProgDaMem;
        }
}

void control::do_condExe() {
        bool      execFlag;  //Execute Flag
        bool      exitFlag;

        if ((cond==COND_AL) || ((cond==COND_EQ) && (zFlag==1)) || ((cond==COND_NE) && (zFlag==0))) {
                execFlag = 1;
        }

        flush = branchFlag & execFlag;

        wbEn = (execFlag && ~refill) ? eWbEn : 0;
        srWbEn = (execFlag && ~refill) ? eSrWbEn : 0;
        IAregCtl = (execFlag && ~refill) ? eIAregCtl : 0;
        dOutCtl = (execFlag && ~refill) ? eDOutCtl : 0;

        exitFlag = (execFlag && ~refill) ? eExitFlag : 0;
}
```

```
/*
 * Debug: Debug Information for ICS_RISC(header file for debug)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(cklm@student.ecu.edu.au)
 * File name: debug.h
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "systemc.h"
#include "def.h"

SC_MODULE(debug) {
```

```
        sc_in<sc_uint<4> >    instId;
        sc_in<sc_uint<4> >    aluCtl;
        sc_out<int>           instIdText;
        sc_out<int>           aluText;

        void do_debug();

        SC_CTOR(debug) {
                SC_METHOD(do_debug);
                sensitive << instId << aluCtl;
#ifdef SIM
                instIdText.initialize(0);
                aluText.initialize(0);
#endif
        }
};


/*
 * Debug: Debug Information for ICS_RISC(source file for debug)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: debug.cpp
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "debug.h"

#define ALUIS       0;
#define ALUIL       1;
#define ALUR        2;
#define ALULB       3;
#define SHRO        4;
#define LOAD        5;
#define STORE       6;
#define BRANCH      7;
#define MUL         8;
#define PECON       9;
#define DMA         10;

void debug::do_debug() {
        sc_uint<4> instIdTmp;
        sc_uint<4> aluCtlTmp;
        instIdTmp = instId.read();
        aluCtlTmp = aluCtl.read();

        switch (instIdTmp) {
                case INST_ALUIS     : instIdText = ALUIS;  printf("ALUIS \n");    break;
                case INST_ALUIL : instIdText = ALUIL;      printf("ALUIL \n");    break;
                case INST_ALUR      : instIdText = ALUR;   printf("ALUR \n");     break;
                case INST_ALULB : instIdText = ALULB;      printf("ALULB \n");    break;
                case INST_SHRO      : instIdText = SHRO;   printf("SHRO \n");     break;
                case INST_LOAD   : instIdText = LOAD;      printf("LOAD \n");     break;
                case INST_STORE : instIdText = STORE;      printf("STORE \n");    break;
                case INST_BRANCH: instIdText = BRANCH;printf("BRANCH \n");        break;
                case INST_MUL       : instIdText = MUL;    printf("MUL \n");      break;
                case INST_PECON : instIdText = PECON;      printf("PECON \n");    break;
                case INST_DMA       : instIdText = DMA;    printf("DMA \n");      break;
                default: printf("Not Defined Instruction \n");                    break;
        }
}
```

```
/*
 * BusCtl: I/O Bus Control for ICS_RISC(header file for busCtl)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: busCtl.h
```

```
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "systemc.h"

SC_MODULE(busCtl) {
        sc_in<bool>         nRW;
        sc_in<sc_uint<32> > dataOut;

        sc_out<sc_uint<32> > dataIn;
        sc_inout_rv<32>      data;

        void do_busCtl();

        SC_CTOR(busCtl) {
                SC_METHOD(do_busCtl);
                sensitive << nRW << dataOut;
        }
};

/*
 * BusCtl: I/O Bus Control for ICS_RISC(source file for busCtl)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: busCtl.cpp
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "busCtl.h"

void busCtl::do_busCtl() {
        dataIn = sc_uint<32> (data);
        if (nRW) {
                data = sc_lv<32> (dataOut);
        } else {
                data = "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        }
}
```

```
/*
 * ICS_RISC: Top module for ICS_RISC(header file for ICS_RISC)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: ICS_RISC.h
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "systemc.h"
#include "datapath.h"
#include "control.h"
#include "busCtl.h"

SC_MODULE(ICS_RISC) {
        sc_in<bool>                 clock;
        sc_in<bool>                 reset;
        sc_in<sc_uint<32> >         iData;    //Instruction Data

        sc_out<bool>                nRW;
        sc_out<sc_uint<32> >        iAddr;    //Instruction Address
        sc_out<sc_uint<32> >        dAddr;    //Data Address;
        sc_inout_rv<32>             dData;    //Data Bus;

        //Extended Output
        sc_out<sc_uint<3> >         PEOp;     //PE Execution Operation
```

```
sc_out<sc_uint<2> >     PEOpmode;          //PE Operation Mode
sc_out<sc_uint<2> >     PEConfig;          //PE Configuration Mode
sc_out<sc_uint<4> >     PESel;             //PE Selection
sc_out<sc_uint<3> >     DMAOp;             //DMA Operation Selection
sc_out<bool>            DFBSel;            //DFB Selection (2 Sets)
sc_out<sc_uint<4> >     dataAmt;   //Amount Data to Transfer
sc_out<sc_uint<6> >     startAddrDFB;      //Start Address of DFB
sc_out<bool>            SRAMRegSel;                //SRAM/ICS_RISC Reg Sel
sc_out<sc_uint<5> >     startAddrSRAMReg;  //Start Address of SRAM/ICS_RISC Reg
sc_out<bool>            memSel;            //Memory Selection(Program/Data)
sc_out<sc_uint<10> >  startAddrProgDaMem;          //Start Address of Program/Data Mem

sc_signal<sc_uint<32> >     imm;
sc_signal<bool>             cmpFlag;
sc_signal<bool>             srOEn;
sc_signal<bool>             srWbEn;
sc_signal<sc_uint<4> >      aluCtl;
sc_signal<sc_uint<3> >      shiftCtl;
sc_signal<bool>             aluOEn;
sc_signal<bool>             shiftOEn;
sc_signal<bool>             mulOEn;
sc_signal<sc_uint<5> >      opAIdx;
sc_signal<sc_uint<5> >      opBIdx;
sc_signal<bool>             rdAOEn;
sc_signal<bool>             rdBOEn;
sc_signal<sc_uint<5> >      wbIdx;
sc_signal<bool>             wbEn;
sc_signal<bool>             immOEn;
sc_signal<bool>             iAregCtl;
sc_signal<bool>             dAregCtl;
sc_signal<bool>             dInCtl;
sc_signal<bool>             dOutCtl;
sc_signal<sc_uint<32> >     dIn;
sc_signal<sc_uint<32> >     dOut;
sc_signal<bool>             zFlag;
sc_signal<bool>             lbEn;
sc_signal<bool>             lbRWEn;
sc_signal<bool>             tnRW;

void do_OutCtl();

busCtl*    ibusCtl;
control*   icontrol;
datapath*  idatapath;

SC_CTOR(ICS_RISC) {
        ibusCtl=new busCtl("busCtl");
        ibusCtl->nRW(tnRW);          ibusCtl->dataOut(dOut);          ibusCtl->dataIn(dIn);
        ibusCtl->data(dData);
        icontrol=new control("control");
        icontrol->clock(clock);      icontrol->reset(reset);          icontrol->dIn(lData);
        icontrol->zFlag(zFlag);      icontrol->imm(imm);              icontrol->cmpFlag(cmpFlag);
        icontrol->srOEn(srOEn);      icontrol->srWbEn(srWbEn);        icontrol->aluCtl(aluCtl);
        icontrol->aluOEn(aluOEn);    icontrol->shiftCtl(shiftCtl);    icontrol->shiftOEn(shiftOEn);
        icontrol->mulOEn(mulOEn);    icontrol->opAIdx(opAIdx);        icontrol->opBIdx(opBIdx);
        icontrol->rdAOEn(rdAOEn);    icontrol->rdBOEn(rdBOEn);        icontrol->wbIdx(wbIdx);
        icontrol->wbEn(wbEn);        icontrol->immOEn(immOEn);        icontrol->iAregCtl(iAregCtl);
        icontrol->dAregCtl(dAregCtl);icontrol->dInCtl(dInCtl);        icontrol->dOutCtl(dOutCtl);
        icontrol->lbEn(lbEn);        icontrol->lbRWEn(lbRWEn);
        icontrol->PEOp(PEOp);        icontrol->PEOpmode(PEOpmode);    icontrol-
>PEConfig(PEConfig);
        icontrol->PESel(PESel);      icontrol->DMAOp(DMAOp);          icontrol-
>DFBSel(DFBSel);
        icontrol->dataAmt(dataAmt);  icontrol->startAddrDFB(startAddrDFB);
        icontrol->SRAMRegSel(SRAMRegSel);    icontrol->startAddrSRAMReg(startAddrSRAMReg);
        icontrol->memSel(memSel);    icontrol->startAddrProgDaMem(startAddrProgDaMem);
        idatapath=new datapath("datapath");
        idatapath->clock(clock);     idatapath->reset(reset);         idatapath->imm(imm);
        idatapath->cmpFlag(cmpFlag); idatapath->srOEn(srOEn);         idatapath-
>srWbEn(srWbEn);
```

```
                idatapath->aluCtl(aluCtl);          idatapath->aluOEn(aluOEn);          idatapath->shiftCtl(shiftCtl);
                idatapath->shiftOEn(shiftOEn);      idatapath->mulOEn(mulOEn);
                idatapath->opAIdx(opAIdx);          idatapath->opBIdx(opBIdx);          idatapath-
>rdAOEn(rdAOEn);
                idatapath->rdBOEn(rdBOEn);          idatapath->wbIdx(wbIdx);            idatapath->wbEn(wbEn);
                idatapath->immOEn(immOEn);          idatapath->iAregCtl(iAregCtl);      idatapath-
>dAregCtl(dAregCtl);
                idatapath->dInCtl(dInCtl);          idatapath->dOutCtl(dOutCtl);        idatapath->dIn(dIn);
                idatapath->lbEn(lbEn);              idatapath->lbRWEn(lbRWEn);          idatapath->zFlag(zFlag);
                idatapath->iAddr(iAddr);            idatapath->dAddr(dAddr);            idatapath->dOut(dOut);

                SC_METHOD(do_OutCtl);
                sensitive << clock.pos();

        }
};

/*
 * ICS_RISC: Top module for ICS_RISC(source file for ICS_RISC)
 * Copyright(c) 2005 by Chul KIM, All right reserved
 * Author: Chul KIM(ckim@student.ecu.edu.au)
 * File name: ICS_RISC.cpp
 * Revision history: Version1
 * Date: 5/5/2005
 */

#include "ICS_RISC.h"

void ICS_RISC::do_OutCtl() {
        tnRW = dOutCtl;
        nRW = tnRW;
}
```